AMIGA assembler

Peter Wollschlaeger

• MOLTISSIMI ESEMPI PRATICI • ELENCHI DELLE ROUTINE DI SISTEMA • ISTRUZIONI PER COSTRUIRE ROUTINE ASSEMBLER COLLEGABILI AL BASIC AMIGA

CONTIENE DISCO 3½"

GRUPPO EDITORIALE JACKSON

AMIGA assembler

Peter
Wollschlaeger

GRUPPO
EDITORIALE
JACKSON
Via Rosellini, 12
20124 Milano

Indice

PREFAZIONE

Io stesso suo tempo ho imparato l’assembler su un IBM 360. A quell'epoca i|tempo di calcolo era esageratamente caro, questo è il motivo per cui noi principianti siamo stati vessati con moltissima teoria prima di poter caricare il costosissimo computer con i nostri primi semplici programmi. Durante le molte ore di teoria mi sono sempre dovuto porre dei problemi su cosa era giusto e cosa no. Ho cominciato a capirci veramente qualcosa solo molto tempo dopo, cioè nella pratica. Purtroppo ancora al giorno d’oggi s'incontra in molti libri lo stesso sistema di procedere. Talvolta il primo programma arriva dopo centinaia di pagine di teoria nuda e crude o addirittura, se siamo sfortunati nel secondo volume.

Nel presente testo voglio procedere in maniera diversa.

La teoria dovrà arrivare fino al punto in cui è assolutamente necessaria per la comprensione dei primi semplici programmi. Nel nostro caso siamo arrivati a ciò solo nel quarto capitolo, ma completamente senza basi non è possibile andare avanti. Quando il primo programma gira, leggeremo altra teoria fino al programma successivo, che ovviamente è un po’ più complicato. E' così aumenteremo nella difficoltà finché alla fine non saremo in grado di scrivere da soli anche dei programmi complicati.

Ancora qualcosa: un Assembler è sempre relativo a una CPU ben determinata, nel nostro caso il 68000. Questa CPU a dir la verità è presente anche nell'Atari ST e nel Macintosh, ma un programma per Atari non potrà funzionare sull'Amiga.

Per questo motivo rinuncio a una grossa cerchia di lettori e mi accingo a scrivere questo testo solo per l'Amiga. Tuttavia, se mai passerete a un altro 68000, potrete portare con voi quello che avete appreso qui; sarà necessario solo imparare a conoscere le parti interne del sistema operativo dell'altro 68000.
Non è possibile programmare in Assembler senza solide conoscenze di Exec, DOS o Intuition. Quindi questi argomenti non costituiscono solo un capitolo nel presente libro, ma sono in pratica un filo rosso che si protrae per tutti i capitoli.

Infine un consiglio: il mancato funzionamento di un programma è sempre causato dalle solite piccolezze. Purtroppo però in Assembler anche un piccolissimo errore viene punito con lunghi tempi di attesa, in quanto, dopo l'eliminazione dell'errore, non è sufficiente un semplice RUN, bensì è necessaria una intera rielaborazione.

Cerchiamo comunque di non scoraggiarci! Per ogni errore impariamo cose nuove e alla fine faremo sempre meno errori. E' comunque necessaria una certa resistenza.
D'altra parte è anche chiaro che chi ha imparato un linguaggio di programmazione, il BASIC o qualcos'altro, imparerà più facilmente un secondo linguaggio.
In linea di massima l'Assembler non è più difficile del BASIC, e solo un po' più complesso. Di tale complessità fa parte anche il fatto che l'utente (contrariamente per quanto avviene per il BASIC) deve sapere come funziona il computer.

D'altra parte però è anche divertente poter programmare direttamente il proprio computer, in un linguaggio di programmazione evoluto si dipende sempre dal compilatore o dall'interprete. Se uno di questi due funziona male o è troppo lento la questione rimane irrisolta, mentre in Assembler sarà possibile a tal punto affrontare il problema direttamente.

Peter Wollschlaeger

A chi è destinato il presente testo?

Il presente testo si rivolge ai principianti e a chi proviene da altri sistemi. Questi ultimi possono saltare le sezioni da 1.1 a 1.7 del Capitolo 1 e le sezioni 2.1 e 2.2 del Capitolo 2.

Purtroppo devo deludere coloro che provengono da altri sistemi, in particolare se provengono da sistemi a “8 Bit”.

Le conoscenze di cui essi dispongono relativamente allo Z80, al’8088 oppure al 6502 servono purtroppo molto poco, o addirittura potrebbero dare fastidio. In effetti, come programmatori per lo Z80 oppure per il 6502 ci si abitua a determinate tecniche e a determinati modi di pensare che in verità sarebbero applicabili al 68000 ma che darebbero inutilmente origine a programmi esageratamente lunghi. Dimentichiamo quindi tutti i tipi di indirizzamento che abbiamo appreso, cancelliamo completamente il concetto di Accumulatore, Banking e Paging e molte altre cose: meglio sarebbe se dimenticassimo tutto!

Chi ha già esperienza relativamente al 68000 potrà cominciare dal Capitolo 4. Lo stesso vale anche per il lettore che proviene dai Mini, in particolare dal VAX.

A essere sinceri, il 68000 è un processore sulle cui fantastiche proprietà si potrebbero scrivere chilometriche tesi di laurea per i laureandi in informatica, ma non è quello che sto per fare. Nei Capitoli da 1 a 3 pongo solo le basi che è necessario conoscere al fine di poter scrivere i primi programmi. Dopo, ci sono solo esercizi, esercizi, esercizi. Solo nei Capitoli 10 e 11 cominciano gli apprezzamenti del 68000, poi di nuovo esercizi.

Durante tali primi capitoli lo schema applicato è il seguente:

1. Determinazione dei compiti
2. Rappresentazione dei comandi necessarie delle funzioni TOS
3. I listati di programma
4. Spiegazione dei listati

In alcuni punti questo ordinamento viene interrotto, perché talvolta è più utile chiarire il punto 2 contemporaneamente con il listato.

In ogni caso non bisognerà mai leggere il listato per primo: la prima volta sarà opportuno saltarlo.

Naturalmente sarebbe molto difficile annotarsi tutte le informazioni di tutti i Capitoli.
Di conseguenza nell'appendice vengono riassunti, fra l'altro, i comandi del 68000 e le funzioni DOS in un formato compatto. In tal modo si rende possibile una ricerca veloce.

Gli utilizzatori dell'Assembler SEKA facciano particolare attenzione al Capitolo 11 e alle appendici. In esse troveranno le tabelle LVO e altri dati importanti che, nel caso di altri Assembler, sono disponibili direttamente sul dischetto.

Chi non ha familiarità con il CLI legga immediatamente l'appendice 5 prima del Capitolo 4!

CAPITOLO 1

Assembler:

Cos’è l'Assembler?
Come si programma in Assembler?
Quando è necessario l'Assembler?
Di che Software si ha bisogno?

Nel presente Capitolo si mostra prima di tutto che cosa è l'Assembler, quando si ha bisogno di lui e di che tipo di Software si ha bisogno al fine di poter scrivere un programma in Assembler.

Ancora una cosa: se scoprite un paio di frasi in linguaggio oscuro, che non comprendete, continuate a leggere normalmente, Quando ne avrete veramente bisogno vi verranno spiegate.

1.1 Livello basso: Linguaggio macchina

Un computer in sé e per sé è molto stupido, è solo incredibilmente diligente. Si è soliti dire che solo chi è molto stupido può essere tanto diligente. In effetti questa macchina non riesce nemmeno a contare fino a 3 e nemmeno fino a 2, conosce semplicemente lo 0 e l'1. La causa di ciò è che i circuiti elettrici di cui un computer è composto possono accettare solo due stati, cioè tensione presente o tensione assente, corrente che passa o che non passa, un transistor che conduce o che è interdetto. Alcune centinaia di migliaia di questi circuiti (transistor) costituiscono la CPU (Central Processing Unit, Unità Centrale, praticamente il cuore del computer) mentre più di otto milioni di essi (nel caso in cui si disponga di un mega-Amiga) costituiscono la memoria del computer.

Un programma non è nient'altro che un determinato stato di queste memorie. Dal momento che sarebbe altamente poco pratico scrivere un programma nella seguente maniera "transistor 1 conduce - transistor 2 pure - transistor 3 interdetto – transistor 4 conduce ecc." si è arrivati velocemente a una specie di stenografia di questo genere: si è convenuto che uno di questi stati corrisponda a 0 e l'altro a 1. In questo modo si potrà scrivere un programma in una maniera molto più compatta. per es, come segue:

```
010111001101010101010 Etc.
```

Non vi piace? Ok. questo è il linguaggio macchina, nient'altro!!!!!

Suppongo che abbiate già riconosciuto che questo campione di 0101011 non è nient'altro che un numero in notazione binaria (ne riparleremo in seguito). Questi numeri possono venire conveniti in numeri decimali oppure esadecimali, cosa che risparmia un po' di carta, ma rimane comunque linguaggio macchina.

1.2 Più in alto: Assembler

Molti credono ancora che l'Assembler sia questo linguaggio macchina. Grazie al cielo si sbagliano: l'Assembler è lo stadio immediatamente successivo ad esso ed è stato a suo tempo un grandissimo progresso, rimanendo per molti anni l'unico linguaggio in assoluto.

Bit e Byte
Un bit è una posizione di memoria. un circuito in un computer che può accettare solo questi stati di 0 o di 1. Per motivi tecnici sono sempre raccolti insieme 8 bit alla volta e questi 8 bit sono chiamati byte. La memoria di un computer è composta da migliaia o milioni di byte. Al fine di potersi rivolgere ad ogni singolo byte, li si ha numerati. Questi numeri "civici" dei byte vengono chiamati indirizzi, con gli 8 bit di un byte è possibile scrivere in binario i numeri da 00000000 fino a 11111111 che in decimale è da 0 a 255. In un byte (l'esperto direbbe su un indirizzo) posso scrivere un numero di questo genere e in seguito rileggerlo. Le cosiddette apparecchiature periferiche come il monitor, la tastiera, o una stampante sono collegate con una parte della memoria (questo byte). Se io scrivo un numero all'indirizzo giusto, questo provoca un effetto sul monitor, se io leggo qualcosa da un altro indirizzo, questo può essere per es. un tasto della tastiera.

Il movimento è tutto.

Di conseguenza un programma è composto principalmente dalla scrittura di numeri (chiamati anche dati) su di un indirizzo, dalla lettura di altri numeri da altri indirizzi ed essenzialmente dalla copiatura di dati da un indirizzo (per es. tastiera) su un altro indirizzo (per es. monitor). Oltre ai dati, un computer conosce anche i comandi, naturalmente sotto forma di 010101110, cioè sotto forma di numeri.

Ipotizziamo che il numero 11111111 sia il comando "copia" e che noi vogliamo copiare dei dati dall'indirizzo 0000011 (in decimale) all'indirizzo 00001001 (in decimale 9); questo programma in linguaggio macchina sarà

```
11111111
00000011
00001001
```

in Assembler al contrario si scriverà

```
MOVE 3,9
```

Move significa spostare; in questo caso il comando è: sposta ciò che si trova nel byte con l'indirizzo 3 al byte con l'indirizzo 9. Al fine di evitare immediatamente un grosso errore, diciamo subito che il byte 3 resta invariato e che viene solo copiato nel byte 9. Mi si farà osservare giustamente che il comando dovrebbe essere più precisamente COPY, ma qui si chiama MOVE.

A questo punto abbiamo imparato la differenza fra l'Assembler e il linguaggio macchina; è già un progresso, no?

E già abbiamo il problema successivo. Il concetto di Assembler ha in effetti un significato doppio: da un lato si tratta del linguaggio di programmazione, come per es. il BASIC oppure il PASCAL; l'unica differenza come abbiamo già visto è data dal fatto che l'Assembler è sempre relativo ad una determinata CPU. Esiste per es. l'Assembler per Z80, l'Assembler per l'8088 e naturalmente l'Assembler per il 68000, di cui stiamo trattando nel presente testo. Il linguaggio ha dei comandi, come anche gli altri linguaggi, che l'utente dovrà battere sulla tastiera.

La grande differenza rispetto per es. al BASIC sta solo nel fatto che alla fine non si deve solo battere RUN, ma che il testo deve prima venire assemblato. Ciò viene eseguito da un programma che si chiama assemblatore. Questo programma traduce il testo in linguaggio macchina, cioè nella sequenza di 0101010, che la CPU è in grado di comprendere.

1.3 Molto più: in alto: linguaggi evoluti

In un linguaggio evoluto, come per es. il PASCAL, è possibile immettere anche del testo; anche questo deve venire tradotto, solo che il programma di traduzione non si chiama Assemblatore bensì Compilatore. Ciò significa che dopo un procedimento di assemblaggio oppure dopo un procedimento di compilazione, deriva un programma in linguaggio macchina, che può venire eseguito su di un computer. Non occupiamoci ora né della dimensione né della velocità dei programmi.

Per un interprete, la cosa è completamente diversa; il rappresentante più tipico di questo genere è certamente il BASIC. Anche qui il programma viene inserito come testo. Forse, dopo l'immissione, viene ancora un pochino elaborato e compresso, ma resta comunque testo, che non è neanche lontanamente parente con il linguaggio macchina. Di conseguenza il computer non è in grado di eseguire un programma in BASIC.

Questo compito viene preso in carico da un interprete. Esso legge il testo in BASIC carattere per carattere e lo confronta con i comandi in BASIC. Nel caso in cui esso trovi un comando in BASIC, chiama immediatamente una routine che esegue tale comando. La routine si trova naturalmente nella memoria sotto forma di programma in linguaggio macchina eseguibile. Essa si occupa anche di cercare i dati appartenenti ad un determinato comando in BASIC (parametri). Naturalmente anche l'interprete è un programma in linguaggio macchina. Tutti gli interpreti di BASIC veloci sono scritti in Assembler.

1.4 Principi di base dell'Assembler

Cerchiamo di approfondire meglio il fatto che un compilatore produca dei codici in linguaggio macchina esattamente come un assemblatore.

in Pascal per es. è sufficiente scrivere:

```
Write ('Ciao!')
```

e in Assembler invece scriveremo (solo come es.):

```
MOVE #'C',4711
MOVE #'i',4712
MOVE #'a',4713
MOVE #'o',4714
MOVE #'!',4715
```

Quindi un programma in Assembler è la suddivisione per es. di un comando in Pascal, come WRITE, in molti singoli comandi. Si potrebbe anche dire: il Pascal conosce una determinata quantità di comandi, dai quali il compilatore produce la sequenza adatta di comandi in Assembler.

Ogni programma in Assembler (nella forma di testo non ancora tradotta) è sempre più lungo del suo equivalente in un linguaggio evoluto. E' solo dopo l'assemblaggio o la compilazione che un programma in Assembler sarà drasticamente più corto e di conseguenza più veloce del suo equivalente. Ciò è dovuto al fatto che nessun compilatore può generare un codice talmente compatto come lo può fare un programmatore in Assembler. Quest'ultimo in effetti sa che cosa vuole e quindi può "tagliare su misura" ogni sequenza di comandi, mentre un compilatore deve introdurre delle soluzioni universali.

La differenza in velocità rispetto ad un interprete di BASIC è assolutamente drastica, in quanto l'interprete traduce (come già detto) mentre il programma gira, quindi lavora sempre solo su un comando. Ciò significa che quando in un programma un comando viene ripetuto 100 volte, esso verrà anche tradotto 100 volte. In un programma in Assembler, al contrario, il comando è già tradotto.

1.5 Quando Assembler e quando no?

Probabilmente la velocità può non essere importante per alcune persone, ma ci sono altri motivi.

Un interprete in BASIC (o un compilatore in Pascal) graffia solo sulla superficie del gigantesco potenziale di possibilità contenute in un computer. Se si vuole qualcosa di più o qualcosa di diverso sarà sufficiente dirlo alla CPU, tuttavia nella sua lingua che è l'Assembler.

Ancora un motivo: si dovrebbe utilizzare sempre il linguaggio che risolve il problema in questione con lo sforzo minimo. Spesso, se non sempre, non si tratta dell'Assembler. Arriverei quasi a dire che meglio si conosce l'Assembler, meno lo si usa. Un programmatore in Assembler sa infatti che cosa sta comandando alla CPU con quale comando per es. in BASIC, e perviene di conseguenza a programmi migliori. Devo comunque evidenziare ancora una volta che l'Assembler pone le basi per delle ottime conoscenze del funzionamento di un computer.

Quindi consoliamoci: è solo imparando bene l'Assembler che si possono acquisire tali conoscenze.

Se si analizza un programma che è stato scritto o deve venire scritto in un linguaggio evoluto si rileva immediatamente che il problema della velocità è presente solo in alcuni punti (o addirittura solo in un punto) oppure che manca la funzione adeguata. Basterà quindi scrivere solo questa parte in Assembler e collegarla al linguaggio evoluto. Impareremo in seguito come fare ciò.

Ripetiamo quindi: i linguaggi molto lontani dal linguaggio macchina sono chiamati linguaggi evoluti. L'Assembler non appartiene a questa categoria di linguaggi, ma ci permette di acquisire una solida formazione di base.

1.6 Di quale Software si ha bisogno

Le sequenze di lavoro tipiche dello sviluppo di un programma in Assembler sono l'immissione del testo l'assemblaggio. Il linkaggio (vedremo in seguito di cosa si tratta) e il test.

Si tratta degli strumenti e, come in ogni lavoro, per lavorare bene occorre possedere gli strumenti giusti. Ci troviamo infatti in presenza di un'offerta molto ampia e i cataloghi dei produttori promettono tutti molto. Vorrei dare alcuni consigli, da tenere presenti al momento della scelta, presentando alcuni prodotti tipici. Una cosa non dovrà venire dimenticata: un Assembler è un utensile per professionisti che ha bisogno di una buona documentazione e di supporto. Naturalmente è possibile far girare, con tentativi ed

errori, anche un gioco copiato. Per un Assembler, lo stesso potrebbe valere solo per persone dotate di grande pazienza e con un tempo infinito a disposizione. Potrebbe addirittura avvenire che tutti i programmi contenuti nel presente testo non girino nell'Assembler specifico del lettore, perché il tale Assembler magari ha bisogno di un punto in un determinato campo, di cui il mio non ha bisogno. Perché quindi impazzire se nel manuale c'è tutto?

1.6.1 L'Editor

L'Editor è necessario per l'immissione dei testi e per la correzione, il testo viene chiamato sorgente (Source-Text). Normalmente l'Editor viene venduto insieme con l'Assembler. Sarà possibile tuttavia utilizzare un normale programma di elaborazione dei testi, limitandosi all'immissione di testo puro (nessun carattere di formattazione o di controllo). Anche ED (facente parte dell'Amiga) è utilizzabile a questo scopo.

L'Assembler segnala gli errori indicando un numero di riga, il testo però viene inserito senza numeri di riga. Di conseguenza l'Editor dovrebbe poter essere in grado di eseguire un “Go to-riga”. Naturalmente sarebbe ancora meglio se riuscisse a saltare direttamente alla riga con l'errore. Dal momento che spesso l'utente trasforma per se stesso determinati programmi oppure spesso copia parti di testo (con leggere modifiche) dovrebbe essere inoltre possibile la movimentazione e la copiatura di blocchi.

1.6.2 L'assemblatore

Se il testo è pronto (e su dischetto) lanciare l'Assemblatore, fornendogli il nome del file contenente il testo sorgente (Source-File). L'Assemblatore produce il codice in linguaggio macchina (questi 010101010) (chiamati anche codice oggetto) e lo pone nel file di destinazione (file-oggetto) sul dischetto. Tutto ciò prevede naturalmente perdita di tempo e in questo modo, appare sensato, poter assemblare “in memoria”. Ciò significa, che l'Assemblatore scrive, su richiesta, il codice direttamente nella memoria e con ciò si può iniziare il programma a scopo di prova. Non si dovrà tuttavia sopravvalutare questa caratteristica, poiché lo stesso risultato può venire raggiunto anche con un RAM-Disk o, dal punto di vista del tempo impiegato, anche con un disco rigido. In entrambi i casi l'Assemblatore deve girare naturalmente su un RAM-Disk oppure su un disco rigido. In caso contrario, sono importanti le seguenti caratteristiche:

“File include”:

L'Assemblatore è in grado di collegare dei moduli di testo. Questo è molto importante, poiché i programmi Amiga necessitano sempre delle cosiddette Libraries (Biblioteche) che sono disponibili sotto forma di moduli di testo. Inoltre, dopo un certo periodo di tempo. Il lettore disporrà di una piccola Biblioteca di Routine, che utilizzerà in quasi tutti i programmi.

Capacita di gestire le Macro: le macro vengono trattate dettagliatamente nel Capitolo 6. Per il momento accontentiamoci solo di quanto segue: le macro contribuiscono sostanzialmente alla razionalizzazione del lavoro e aiutano ad evitare gli errori.

Messaggi di errore: Consultare il manuale. Più è lunga la lista dei messaggi di errore, meglio verremo informati.

Segnalazioni: un buon Assembler segnala (consiglia) quando non si è programmato in maniera ottimale. Anche qui vale quanto detto precedentemente: più segnalazioni ci sono, meglio è.

1.6.3 Il Linker

A questo punto abbiamo bisogno del Linker che ha due compiti:

Sappiamo di poter suddividere un programma in moduli, che in seguito assembleremo e verificheremo separatamente (molto consigliabile in caso di programmi lunghi).
Questi moduli dovranno quindi venire legati insieme dal Linker per formare un programma. Un altro motivo per l'uso del Linker si trova all'interno dell'Amiga stesso. Ogni programma possiede una intestazione (in Inglese Header), nella quale si trova per es. l'indicazione di quanto è lungo il programma. Senza questa informazione l'Amiga non può né caricare né far partire il programma. Di conseguenza il Linker deve collegare per lo meno questa intestazione al programma stesso. Ci sono anche degli assemblatori che fanno la stessa cosa e ciò risparmia l'utilizzo del Linker, cosa che è senza dubbio positiva. Infatti, particolarmente l'ALINK (il Linker standard) è molto lento. Lo svantaggio della mancata modularizzazione potrebbe venire compensata dalla capacita di "Include" e di gestione delle macro.

1.6.4 Il Debugger

Se adesso lanciamo il programma, abbiamo tre possibilità: che il programma giri, che il programma non giri, o che funzioni male. Al fine di trovare il Bug, abbiamo diverse possibilità. La più semplice (e di solito quella che è coronata da maggior successo) è un'analisi approfondita del testo sorgente combinata con una intensa riflessione.

Se però vogliamo sapere che cosa fa il programma in un determinato punto, oppure quali valori hanno alcune variabili, la cosa diventa più difficile. Senza dubbio è possibile inserire in tali punti per es. un "PRINT AB" cosa che però in Assembler è abbastanza complessa, come vedremo in seguito (infatti non esiste nessun comando di Print).
Quindi è molto più pratico utilizzare un cosiddetto Debugger. Si tratta di un programma con il quale faremo girare il nostro programma in passi singoli e tramite il quale potremo vedere in ogni punto i valori delle variabili.

Non aspettiamoci comunque troppo da un Debugger. Un programma di questo genere non è assolutamente facile da far funzionare e nella fase iniziale, ci procurerà più grattacapi di quanto non ci aiuterà. A ciò si aggiunge il fatto che i principianti di solito fanno degli errori che vengono rilevati già dal compilatore. Ripetiamo ancora una volta, visto che è cosi importante: l'errore è sempre nel testo sorgente. Ecco perché un'analisi approfondita e una riflessione intensa costituiscono il Debugger migliore. Quando ci procureremo un Debugger, dovremo fare attenzione principalmente a 2 cose:

prima di tutto dovrebbe trattarsi di un Debugger simbolico. Significa: in un programma in Assembler non si lavora mai con indirizzi assoluti, bensì con delle Labels (etichette) che sono gli indirizzi simbolici. L'Assembler tiene una tabella, nella quale annota i numeri reali in corrispondenza dei "simboli". Un Debugger simbolico interviene semplicemente su tale tabella e da ciò deriva la seconda esigenza, e cioè che il Debugger sia compatibile con l'Assembler.

1.7 Cosa si dovrebbe acquistare (e cosa no)

L'Editor, il Compilatore, il Linker e il Debugger (se disponibile) vengono di solito offerti tutti insieme in un pacchetto. Spesso, allegato ad essi, si trova anche il cosiddetto Shell (area propria dell'utente) che rende possibile passare direttamente per es. dall'Editor allo Shell, da dove poi si potrà chiamare il Linker. Ciò significa, se si lavora per es. senza RAM-Disk, che si potrà procedere molto più velocemente senza passare per il Workbench o il CLI. Inoltre un buon Shell ha anche il vantaggio di essere in pratica un Workbench "tagliato su misura" per la programmazione.

1.8 Tre Assembler a confronto

La seguente sezione è una relazione di testo che io ho già pubblicato in "Computer persönlich". In essa vengono affrontati degli aspetti che verranno spiegati più chiaramente solo più avanti nel presente testo. Cerchiamo quindi di non farci confondere le idee, nella peggiore delle ipotesi ignoriamo pure questa sezione. Sono sicuro, nonostante tutto, che il lettore alla fine della presente sezione sarà comunque in grado di sapere come i singoli Assembler fanno fronte alle sue esigenze.

Sono stati provati gli Assembler della Metacomco, Kuma (K-SEKA) e HiSoft (DEVPAC Amiga). L'Assembler della Metacomco è l'Assembler standard per l'Amiga. Ciò tuttavia non rappresenta un motivo per scegliere proprio questo, dal momento che i concorrenti dovranno pure offrire qualcosa che li differenzi in maniera vantaggiosa dallo standard.

Rimaniamo per il primo momento con l'Assembler della Metacomco, cosa che ci permetterà in seguito di sottolineare meglio le differenze. Viene fornito un dischetto con la scritta “Macro Assembler for the Amiga, Version 11.00”. Ovviamente non si ratta della versione 11 dell'Amiga. Il produttore fornisce da anni un Assembler per il 68K e lo adatta ai diversi computer.

Alla lettura del manuale comincia la frustrazione. All'infuori del titolo, in tutto il manuale non si incontra più la parola Amiga, e ciò significa che neanche mezza frase spiegherà come far girare un programma in Assembler su di un Amiga.

Metacomco poco documentato

Resta quindi compito del lettore cominciare a provare direttamente sul dischetto, dove troverà un semplice esempio, che tuttavia lascia completamente inutilizzate le capacità dell'Amiga (grafica, Multitasking). Nel manuale non viene citato minimamente come procedere con le Libraries che sono molto importanti per l'Amiga, e le questioni come la differenza fra le routine CLI, i compiti CLI e i compiti di Intuition restano assolutamente escluse. In pratica:chi non è già completamente padrone dell'Assembler per il 68000 in generale, e non conosce l'Amiga in particolare, non ha nessuna possibilità, tramite questo manuale, di imparare la programmazione dell'Amiga. Inoltre la Metacomco da per scontato che l'utente disponga di una documentazione completa per l'Amiga (o di un libro come questo).

Solo nell'introduzione si raccomanda di procurarsi un manuale per l’utente DOS nonché un manuale per il programmatore. E stupisce ancora di più che le prime 18 pagine del manuale dell’Assembler si occupano dell’Editor ED, che è già descritto nel manuale DOS (la Metacomco non fornisce nessun Editor proprio). Rimangono le pagine da 19 a 50, nelle quali viene presentato l'Assembler in sé e per sé. Si tratta di un Assembler standard per il 68K convenzionale, ma con solide basi, che adempie completamente le specifiche Motorola. Le singole direttive vengono elencate in sequenza. Anche qui si dà per scontato che il lettore sappia cosa farsene, infatti mancano gli esempi. Le ultime otto pagine descrivono molto chiaramente le funzioni Macro. Questa sezione che si occupa delle Macro è veramente encomiabile e quasi assolutamente indispensabile per la programmazione nell'Amiga. Nei file Include che vengono consegnati insieme, sono contenute moltissime Macro, che d’altra parte si attengono precisamente al listato del manuale Kernel (documentazione Amiga). Alle Macro possono venire attribuiti fino a 36 argomenti (0 … 9, A … Z), laddove però l'argomento numero 0 è sempre riservato al tipo (B, W, L). Le Macro possono anche chiamare delle Macro definite precedentemente. Questo inscatolamento è permesso fino ad una profondità di 10.

All’interno di una Macro possono venire verificate certe condizioni ed eventualmente lasciate delle Macro espansioni.

SEKA: molto lontano dallo standard

L'Assembler SEKA, in teoria, è fantastico, ma la realizzazione in pratica delle idee è insoddisfacente. Il SEKA è un programma, un Editor, un Assemblatore e un Debugger tutti insieme. Tutte queste parti si trovano permanentemente nella RAM, Anche l'Assemblaggio ha luogo in memoria ed è di conseguenza velocissimo. Purtroppo viene fornito un solo file Include (DOS-Lib), cosa che rappresenta il più grosso punto debole del sistema. Mancano i file Include per programmi tipici per Amiga (grafica, suono e Intuition per es.). Se a qualcuno venisse l'idea di procurarsi questi file indifferentemente dalla Metacomco o dalla HiSoft, resterà deluso. Infatti l'Assembler SEKA non conosce nessuna istruzione di Include. I file devono venire introdotti nel testo sorgente, cosa che produrrebbe dei listati lunghissimi. Se si è utilizzatori del SEKA, è consigliabile costruire da soli i file Include sulla base dei listati LVO nell'appendice del presente testo.

Dal momento che questi listati sono orientati alla sintassi standard, sarà necessario comunque apportare due modifiche: il simbolo di sottolineatura prima del nome deve venire tralasciato e dopo il nome devono venire aggiunti due punti.

File Include contengono anche le Macro. Il SEKA in effetti può gestire anche le Macro, ma usa un'altra sintassi. In questo caso il SEKA resta coerente a se stesso, infatti deviazioni notevoli dallo standard sono per lui la regola. Nel Capitolo 4 vengono descritte le differenze sulla base di un programma esempio. Alla fine del listato risulta chiaramente quanto divergano le direttive SEKA dallo standard. Purtroppo ci sono delle differenze anche nella mnemonica per il 68K. Per es. il SEKA non riconosce MOVEA ma insiste ad utilizzare semplicemente MOVE.

Il SEKA viene gestito tramite abbreviazioni con lettere, che permettono un passaggio semplice fra l'Editor, l'Assemblatore e il Debugger. Il punto debole è l'Editor. Concepito originariamente come Editor per righe, è stato ampliato ad un semplice editor per video. Quest'ultimo perciò è organizzato in maniera molto spartana e molto lento, per cui è necessario spesso passare di nuovo al modo riga.

Premendo un tasto si fa partire l'Assemblatore, che lavora con una velocità così elevata, che per un programma medio non ci si accorge nemmeno del tempo di assemblaggio. Se il programma è assemblato senza errori, lo si dovrà opportunamente memorizzare su un disco, in quanto l'esecuzione successiva di Debug potrebbe creare dei problemi. Secondo il manuale è sufficiente porre un Break Point (punto di interruzione) sull'ultimo statement (in questo caso RTS) ma spesso il programma in modo Debug non riesce a raggiungere questo punto. Infatti quando il programma attende un input, esso resta semplicemente sospeso. A questo punto sarà necessario eseguire una nuova partenza. Io resto comunque dell'opinione che le routine di input sono parti essenziali dei programmi. Allora a cosa serve un Debugger, se non collabora?

Il manuale della SEKA, con le sue 34 pagine, è ancora più piccolo di quello della Metacomco, tuttavia il principiante troverà in esso delle informazioni più utili. L'Editor viene descritto nelle pagine da 4 a 7, le pagine da 8 a 13 presentano l'Assembler, quelle da 14 a 18 il Debugger. Dopo altre due pagine sull'I/O di file, viene descritto il Linker alle pagine 21 – 23.

Ho letto due volte il capitolo relativo al Linker, per poter determinare che non si tratta assolutamente di un Linker, bensì dell'Assemblatore che può anche effettuare dei linkaggi. Praticamente c'è un campo per il codice oggetto e uno per il codice da linkare. Con CL si effettua una copiatura nel campo di Link, mentre con RL è possibile leggere un modulo nel campo di Link. L'Assembler produce normalmente dei codici eseguibili, mentre con l'opzione L, esso genera dei codici linkabili.

Ora, con RL, è possibile leggere diversi moduli nel campo di Link (vengono appesi l'uno all'altro). Nel campo del codice non potrà esserci niente, se non il testo in sorgente del primo modulo, che verrà quindi assemblato prima di ogni altra cosa. Se ora si assembla senza l'opzione L, viene linkato il tutto, sempre che si siano scritti tutti i moduli con indirizzamento assoluto. Saremo quindi contenti di avere un Linker al quale dover procurare solo una lista di tutti i file da linkare, dicendogli quindi da quali biblioteche egli dovrà andare a procurarsi i moduli mancanti.

Solo l'HiSoft fornisce un Linker ideale.

Queste caratteristiche sono possedute dal Linker sia della Metacomco che della HiSoft. Il Linker della Metacomco costituisce un Linker standard per l'Amiga. Il linkaggio deve avere luogo con il Linker della Metacomco, cosa che porta via molto tempo. Nel caso della HiSoft è possibile scegliere se l'Assemblatore deve produrre dei codici eseguibili oppure linkabili. Nei casi normali si rinuncerà a questi ultimi. Se si vogliono quindi collegare dei moduli, si ha comunque un vantaggio. L'HiSoft fornisce un Linker che è compatibile con ALINK, solo molto più veloce. Al fine di indicare il progresso effettuato rispetto all'ALINK, questo Linker ha ricevuto il nome BLINK (si pronuncia B-LINK e non BLINK).

E cosi siamo giunti al terzo prodotto, cioè al DEVPAC dell'HiSoft. Dico subito che ho tenuto il boccone migliore per ultimo. Questo pacchetto, oltre al BLINK, è composto dall'Editor/Assembler GENAM, dal Debugger MONAM e da una serie completa di tutti i file I (perfettamente compatibili con Metacomco) nonché di alcuni programmi di dimostrazione. Anche l'importantissimo codice di Startup si trova nel testo sorgente (senza l'errore che è contenuto nel listato della documentazione Amiga). In aggiunta c'è un programma, chiamato GEMINST, che rende possibile una predeterminazione di certi parametri, come per es. dimensioni preferite del Buffer di testo o della griglia di tabulazione.

Nel caso dell'Editor, si tratta di un Editor di video con menu di Pull-down, predisposti per Assembler. Se lo si paragona con Ed, è impressionante la velocità con la quale il

testo possa venir fatto scorrere su video e quella delle funzioni di ricerca e sostituzione.
Il codice di GENAM è molto compatto, con i suoi 33 Kbyte, e di conseguenza viene caricato dal dischetto in maniera sorprendentemente veloce. Questo è già un buon esempio dei vantaggi della programmazione in Assembler sull'Amiga.

Wordstar con Mouse

Si è già scritto molto sui vantaggi e gli svantaggi di Editor comandabili da Mouse, per cui ogni discussione sarebbe inutile in questa sede. Il cursore può venire posizionato tramite i tasti freccia oppure tramite codici di controllo che sono compatibili con il Wordstar, oppure anche con il Mouse. Quest'ultimo è sicuramente molto utile quando si vuole raggiungere una posizione per la quale sarebbe necessario effettuare diversi passaggi.

L'Assemblatore viene chiamato dall'Editor. Anche in questo caso si può estrarre il comando da un menu di Pull-down oppure da una abbreviazione di tasti (Amiga-A).
Dopo il funzionamento dell'Assemblatore ci si trova di nuovo nell'Editor. Nel caso in cui non sia stato rilevato nessun errore, si può abbandonare l'Editor e richiamare il programma, che si troverà, già perfettamente eseguibile, sul dischetto. Si ha comunque ancora l'opzione di produrre un codice linkabile, che potrà venire linkato con ALINK (oppure, ancora più velocemente, con BLINK). Nel caso in cui l'Assemblatore abbia rilevato degli errori, questi verranno visualizzati con un testo per esteso. Dopo averli approfonditi, premendo un tasto, ci si trova di nuovo nell'Editor. A questo punto si potrà saltare alla riga contenente l'errore con un comando di "GO TO riga numero". L'abbreviazione "Amiga-J" (Jump to error) è ancora più veloce e ci conduce direttamente alla prima riga contenente un errore. Inoltre si ha anche l'opzione di assemblare a vuoto, dove è possibile un controllo della sintassi ancora più veloce.

Quanto tempo è necessario prima di poter disporre di un programma eseguibile su dischetto? Scegliamo il listato usato come esempio nel Capitolo 4, utilizzando tuttavia i file Include (disponibili solo con Metacomco e DEVPAC).

Analizziamo prima di tutto i tempi:

SEKA: 5 secondi
Metacomco: 95 (87) secondi
HiSoft: 11 (2) secondi

Il Metacomco spreca molto tempo con il linkaggio, cosa che non accade con gli altri due. Per il SEKA il procedimento di assemblaggio vero e proprio dura solo alcuni decimi di secondo, il resto del tempo viene utilizzato per la memorizzazione sul dischetto. La HiSoft scrive direttamente sul dischetto. Tuttavia non è un problema tenere in RAM-Disk l'Editor/Assembler più diverse Libraries.

Il DEVPAC occupa 33 Kbyte, il MonAm 18 Kbyte, tutte le Libraries insieme (non le si usa quasi mai) occupano esattamente 180 Kbyte, quindi resta sempre spazio a sufficienza per il sorgente e il codice. In questo tipo di funzionamento si hanno praticamente i vantaggi del SEKA con la facilità dei DEVPAC. In questo caso il programma viene originato in circa 1 secondo. Inoltre con il DEVPAC si possono ottenere dei tempi dell'ordine di grandezza dei 2 secondi anche senza RAM-Disk, se si sostituiscono i file Include con istruzioni EQU nel testo, come mostrato dai listati nei Capitoli 4 e 5.

Mon Ami

Il Debugger della HiSoft si chiama MonAm ma io preferisco chiamarlo Mon Ami in quanto si tratta veramente di un Debugger amico. MonAm è un Debugger simbolico, ma lavora senza problemi anche con file codici, senza tabelle dei simboli. MonAm offre tutte le funzioni a monitor, può disassemblare, tracciare, gestire i break point, in pratica è in grado di fare tutto quello che i buoni programmi di questo tipo sono in grado di fare. La cosa sorprendente è che con MonAm si cade sempre in piedi. Ciò è dovuto al fatto che esso intercetta tutti i "Guru" cioè riduce i vettori di eccezione in balia del proprio Handler (gestore). Concettualmente MonAm può far girare un programma (Task) solo se il Task "dorme". Un tentativo di contravvenire a ciò farà apparire la segnalazione "Task must be suspended" (il task deve venire sospeso).

MonAm pone automaticamente un break point al primo comando. Se si vuole, si può attraversare il programma passo passo. In una finestra appariranno i valori di registro, in un'altra un estratto dalla memoria in esadecimale e ASCII. Quindi seguirà un pezzo del testo sorgente con una freccia alla riga corrente. Ciò significa che per ogni comando si potrà leggere direttamente quale effetto esso ha sulle variabili in memoria e sul registro. A questo punto l'identificazione dell'errore sarà inevitabile.

I manuali migliori sono quelli della HiSoft.

Il manuale del DEVPAC è veramente esemplare. Esso comincia con istruzioni precise per la realizzazione delle copie di sicurezza. Quindi affronta come installare il sistema per uno o due drive oppure per il disco rigido. Quindi ha luogo un corso molto veloce nel quale viene spiegato chiaramente, sulla base di un programma campione sul dischetto, come far funzionare l'Editor, l'Assemblatore e il Debugger. A questo punto, sapendo che tutto funziona, ci si può quindi dedicare ai singoli capitoli, i quali descrivono tutto il resto in maniera molto precisa, ben comprensibile e tuttavia compatta.

Nell'appendice vengono descritti alcuni elementi di base, in particolare l'argomento Libraries. che è così importante per l'Amiga. Segue quindi un corso veloce relativo al CLI, seguito da una istruzione molto dettagliata di come installare un disco CLI/DEVPAC quando per es. si passa da WB 1.1 a WB 1.2.

Nel manuale è contenuto ancora un altro libriccino, cioè il “Programming Pocket Reference Guide” (Guida di riferimento tascabile per la programmazione) della Motorola, cioè la documentazione ufficiale di tutti i comandi per il 68000.

Risultato: HiSoft, vincitore ai punti

E' chiaro che per me il pacchetto migliore è il DEVPAC della HiSoft. Il motivo è costituito dalla sua velocità, dal comfort e compatibilità.

Chi vuol provare la programmazione a basso livello nel senso più completo del termine, si trova ben servito dalla SEKA. Con ciò si intende come una volta si programmava il Commodore 64, nel quale la grafica veniva impostata con “Poke, Poke, Poke”, si potrà fare lo stesso con l'Amiga in Assembler, utilizzando "Move, Move, Move”. Rivolgendosi cosi direttamente all'Hardware dell'Amiga, sarà possibile raggiungere delle velocità incredibili, se confrontate con le chiamate di Intuition che funzionano attraverso innumerevoli istanze. Dal momento che queste dovranno venire programmate in maniera interattiva (modifica dell’operando, giudizio dell'effetto) il SEKA sarà l'ideale, in quanto fa dimenticare i tempi di assemblaggio.

Non appena però si dovranno risolvere dei compiti un po’ più complessi, orientati alla strutturazione ed alla razionalizzazione del lavoro, ci si dovrà rivolgere ad un Assembler Classico. Il pacchetto della Metacomco è piuttosto lento e superato, anche se solido. Di conseguenza il DEVPAC rimane l'unica alternativa possibile.

Nel presente testo, per i primi listati, sottolineo le differenze alle quali il lettore dovrà prestare attenzione se lavora con altri Assembler. Tutti gli altri listati sono stati scritti con l'Assembler della HiSoft.

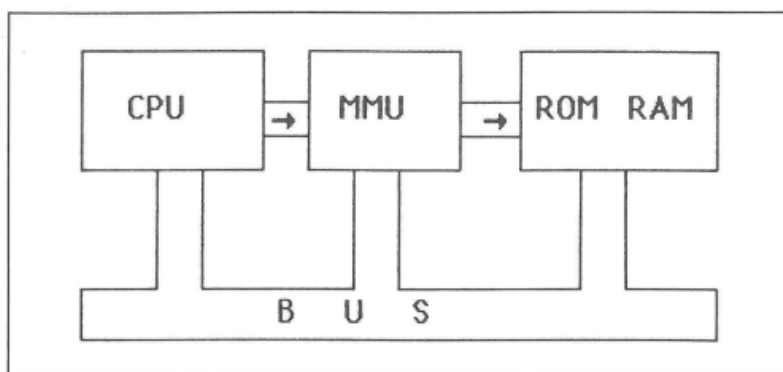
CAPITOLO 2

Composizione di un computer

Registro
Stack

2.1 Il modello di computer

In questo capitolo dobbiamo occuparci anche un po della composizione e della funzione di un computer. Quello che non ci interessa assolutamente sapere in questo momento, è come funziona il sistema elettricamente. Per la programmazione è sufficiente in fatti un cosiddetto modello. E' importante invece sapere che cosa significa e quali sono le prestazioni di CPU, RAM, ROM e Bus. All'interno della CPU, la cosa che ci interessa in particolar modo e il modello dei registri.

Ci troviamo ora nella situazione di chi sta imparando a guidare un'automobile. Analizzeremo che significato hanno lo sterzo, la frizione, il pedale del gas e il freno.

La Fig. 2.1 ci mostra il modello di un computer molto semplificato.

Il componente più importante è la CPU (Central Processing Unit) detta anche processore, cioè il 68000. Esso è responsabile della gestione di tutto il decorso, è in grado di calcolare, di decidere e di paragonare. Da solo, tuttavia, non fa molto: ha bisogno di un programma.

Il secondo componente importante è la memoria, suddivisa nelle parti RAM e ROM. RAM significa letteralmente Random Access Memory, cioè memoria con accesso casuale (è possibile accedere direttamente a qualunque posizione di memoria, e non come per es. in caso di memorizzazione su nastro, solo serialmente) e la stessa caratteristica è posseduta anche dalla ROM. La grande differenza fra le due è che per la RAM è possibile leggere e scrivere, mentre per la ROM (Read Only Memory = memoria di sola lettura) è possibile solo leggere. Una ulteriore differenza: il contenuto della RAM viene perduto nel momento in cui il computer viene spento, il contenuto della ROM, al contrario, è sempre disponibile.

I nostri programmi si troveranno sempre nella RAM, tuttavia utilizzeremo molto anche la ROM (se disponiamo di un Amiga con Kickstart-ROM). Al fine di poter accedere ad una posizione di memoria nella RAM o nella ROM, la CPU deve indirizzare tale posizione di memoria. Questi indirizzi passano attraverso il Bus di indirizzamento.

Il Bus non è nient'altro che un insieme di conduttori, tramite i quali tutte le utenze sono collegate in parallelo. L'espressione Bus viene dal fatto che, se lo si considera in senso figurato, un informazione (per es. un indirizzo) sale alla fermata CPU, si sposta sul Bus e quindi scende alla fermata RAM (oppure ROM). Allo stesso modo i dati (ciò che deve venire posto nei Byte indirizzati/ciò che deve venire letto da tali Byte) si spostano sul Bus dei dati.

La memoria stessa è composta da diversi Chip uguali fra loro. Tutti loro hanno lo stesso piccolo campo di indirizzamento, ma hanno anche un ingresso (Chip Select) tramite il quale è possibile selezionare un determinato Chip. Di conseguenza un indirizzo logico deve venire trasformato in un indirizzo fisico. Ciò viene effettuato dal cosiddetto decodificatore di indirizzi oppure (come nel caso dell'Amiga) da una sua versione ulteriormente migliorata, la cosiddetta MMU (Memory Management Unit).

Per noi è importante sapere che un accesso a campi protetti oppure a indirizzi illegali viene punito con una segnalazione di "Bus-Error".

2.2 Fetch & Execute

Normalmente un programma gira in un computer conformemente al metodo "Fetch & Execute" come dicono gli americani, in italiano ciò significa "Prendi ed Esegui".
La CPU prende un comando dalla memoria e lo esegue. Dopo di ciò: prende automaticamente il comando successivo e lo esegue, ecc., ecc., ecc.. Naturalmente nella memoria ci deve essere qualcosa, che la CPU prende ed esegue, e ciò viene chiamato programma.

2.3 I programmi non sono altro che sequenze di Byte

Un programma non è nient'altro che una sequenza di Byte, che si trova o nella RAM o nella ROM. Naturalmente la CPU non è in grado di sapere dove il programma si trova. Di conseguenza all'avviamento (Reset) si scontra immediatamente, grazie ad una predisposizione Hardware, con una posizione iniziale.
In questo momento la CPU prende sempre una parola (cioè due Byte uno dopo l'altro, quindi 16 Bit) dalla memoria e decodifica tale parola. In essa è senza dubbio contenuto un comando per la CPU.

La CPU elabora questo comando e va a prendere la parola successiva. In un comando possono anche essere contenuti dei dati. In caso di comando di addizione per es., la CPU deve sapere che cosa deve venire aggiunto. Anche l'informazione di quante parole dato fanno parte di un comando è contenuto nella prima parola (la parola di comando).
L'intera memoria è numerata Byte per Byte da 0 alla fine; questi numeri di posizione di memoria sono chiamati indirizzi. La CPU funziona sempre solo con questi indirizzi e tiene internamente un contatore, che punta sempre all'indirizzo attuale. Questo contatore è chiamato “Program Counter" abbreviato con PC.

Ecco un esempio:

Indirizzo (PC)	Comando	Dati
1000	Cancella Parola	
1004	Aggiungi	Operando 1, Operando 2
1010	Return	
1012		

Il listato indica schematicamente un programma che comincia con l'indirizzo 1000.
Al fine di far partire il programma, bisogna impostare il PC a 1000, dopo di che esso comincerà a funzionare.

Il comando 1 occupa gli indirizzi 1000 e 1001. In questo esempio esso ha una parola dato all'indirizzo 1002 e 1003. La CPU elabora questo comando e porta il PC all'indirizzo1004. Del comando 2 (su 1004 e 1005) fanno parte due parole dato (1006-1009), di conseguenza il comando 3 deve partire dall'indirizzo 1010.

Il 68000 conosce comandi senza dati, che quindi sono sempre lunghi una parola,
ma anche comandi contenenti un massimo di 4 parole dato. Ciò significa che, con un 68000, un singolo comando può occupare un massimo di 10 Byte con i suoi dati (5 parole). Come è possibile verificare dal presente schema, ogni comando deve cominciare ad un determinato indirizzo (limite di parola) diversamente sono guai.

Il lettore si chiederà come accade ciò. E' molto semplice. E' possibile (e obbligatorio) modificare il PC. Infatti, se un programma gira non solo comando per comando, ma si è usato per es., anche un GO TO, ciò in Assembler sarà prima di tutto “GO TO indirizzo”. Praticamente ciò per la CPU significa “Imposta il PC = Indirizzo”. Se a questo punto viene fornito un indirizzo sbagliato, il programma viene interrotto.

In pratica questo errore si presenta quando nel programma si definiscono anche dei dati. Se per es. si vuole stampare il testo “Mario Rossi” sarà necessario caricare in un punto qualunque della memoria una sequenza di Byte con il codice ASCII di queste lettere. Se si ha in seguito il testo “20100 Milano” e si vuole stampare questo testo da solo, sarà

necessario sapere quanto è lungo "Mario Rossi". Al fine di risparmiare questo conteggio, i buoni Assembler hanno un comando (EVEN oppure CNOP) che
giustifica i testi (oppure i dati in generale) ad un indirizzo pari. Se l'indirizzo è comunque pari, non succede niente. Un EVEN in più non disturba, uno in meno disturberebbe molto.

Con queste informazioni non dimenticheremo mai di mettere prima di un testo il comando EVEN (o qualcosa di simile) cosa che è comune a tutti i principianti.

2.4 Modo utente e supervisore

Il 68000 conosce due sistemi di funzionamento, chiamati modo-utente e modo-supervisore.
In modo-supervisore girano le routine di base del sistema operativo. I nostri programmi, e principalmente i programmi operativi, girano principalmente in modo-utente.

La cosa importante da sapere è che alcuni comandi per il 68000 sono permessi solo in modo-supervisore. Questi comandi, che nei manuali vengono chiamati comandi privilegiati, non possono venire utilizzati senza essere prima entrati in modo-supervisore.

Diversamente, il programma verrà interrotto con una segnalazione di errore. Il sistema operativo dell'Amiga, in particolare il nucleo Multitasking, reagisce in maniera molto sensibile agli accessi dall'esterno ed è per questo che si dovrà cercare di evitare il modo-supervisore. Ciò non costituisce uno svantaggio. in quanto per particolari caratteristiche del 68000, accessibili solo in modo-supervisore, il sistema operativo ci mette a disposizione delle routine che potremo utilizzare senza problemi.

2.5 Il sistema esadecimale

Il sistema esadecimale è comune nell'Assembler (ed anche molto vantaggioso). Questo è il motivo per cui dobbiamo familiarizzare con tale sistema. Ecco un corso veloce:

La base non è 10, come in un sistema decimale, bensì 16. Per le cifre mancanti da 10 a 15, si scrive da A a F. Per la cifra per es. 345, in decimale si dice 5 unita, 4 decine e 3 centinaia. in esadecimale la base è 16.

La sequenza non sarebbe quindi 1, 10, 100, 1000 bensì 1, 16, 256, 4096.

Sappiamo quindi che F ha il valore 15. Di conseguenza FFFF = 15* 4096 + 15*256+15*16+15*1=65535.

2.6 Un programma in BASIC come esercizio

La Fig. 2.2 riporta un piccolo programma in BASIC per Amiga.

```
While 1
        Input "Un numero n ($n se esa) ";A$
        If left$(A$,1)<>"$" Then
          Print Hex$(A$,Len(A$)-1)
        Else
          A$=Rigth$(A$,Len(A$)-1)
          L=Len(A$)
          X%=0
          For I=L To 1 Step -1
            X%=X%+Val ("&h"+Mid$(A$,I,1))*16^(L-I)
          Next I
          Print X%
         Endif
Wend
```

Fig. 2.2: Conversione da Esa a Decimale in BASIC

Quando si immette un numero decimale, il programma lo trasforma in esadecimale.
Se si immette un numero esadecimale (riconoscibile da $ come primo carattere) si otterrà il suo valore in decimale.

2.7 Il sistema binario

Il sistema binario si trova un gradino più in basso (ancora più vicino al computer). In questo caso la base è 2, per cui in questo sistema sono permesse solo le cifre 0 e 1. La cosa più semplice è tradurre un numero binario in un numero decimale, scrivendo i valori su di esso. Ecco un esempio:

Valore decimale:	32	16	8	4	2	1
Numero binario:	1	0	1	1	0	1

Il risultato sarebbe quindi 32 + 8 + 4 + 1= 45

Anche la trasposizione nel sistema esadecimale è molto facile. Ipotizziamo di avere il seguente numero binario:

```
        1010 0101
```

Vedete subito che l'ho suddiviso in gruppi di 4. Se poi scrivo il loro valore sopra di essi, risulta quanto segue:

```
8 4 2 1   8 4 2 1
-------   -------
1 0 1 0   0 1 0 1
```

Ciò produce (da sinistra) i decimali 10 e 5. In esadecimale, però, per 10 si scrive A, quindi il numero in esadecimale si chiamerebbe A5. Vediamo quindi in Fig. 2.3 un programma in Amiga BASIC per esercitarsi con i numeri binari:

```
WHILE 1
  INPUT "Un numero n (%n se binario) "; a$
  IF LEFT$(a$,1) <> "%" THEN
        x^=VAL(a$)
        F0R i=15 TO 0 STEP -1
         PRINT SGN(x% AND 2^i );
        NEXT : PRINT
  ELSE
    a$=RIGHT$(a$,LEN(a$)-1)
    l=LEN(a$): x%=0
    F0R i=l TO 1 STEP -1
      x%=x%+VAL("&h"+MID$(a$,i,1))*2^(l-i)
    NEXT
    PRINT x%
  END IF
WEND
```

Fig 2.3: Conversione da binario a decimale e ritorno

Se si immette un numero decimale, il programma lo fa uscire in binario. Se si immette un numero binario (riconoscibile da % come primo carattere) ne otterremo il valore in decimale. Il simbolo di percentuale (%) è il prefisso in Assembler per i numeri binari.

2.8 Stack: funzione e compiti

Il programma più corto che è possibile scrivere in Assembler per l'Amiga è:

```
CLR -(SP)
```

e ciò nonostante si è già utilizzato lo Stack. SP (oppure A7, che è la stessa cosa) sarà presente spessissimo in quasi tutti i programmi; un ottimo motivo per analizzarlo immediatamente. Lo Stack è una memoria (un pezzo di RAM) con caratteristiche particolari. Lo si chiama anche LIFO, che sta per “Last in, First Out" oppure anche memoria Stack. I dati vengono immessi nella memoria Stack, come se venissero sovrapposti in una pila. Sarà in seguito possibile prelevare qualcosa solo dall’alto di tale pila (dal ”Top of Stack"). Questo significa che se memorizziamo i dati A, B e C in questa sequenza nello Stack, potremo leggerla solo nella sequenza C, B, A. A questo punto il trucco è solo di far in modo che la CPU non prelevi mai i dati dallo Stack, ma li copi in qualche altra direzione. Ciò viene gestito dal cosiddetto Puntatore allo Stack, abbreviato con SP (Stack-pointer). Un'altra annotazione: lo Stack cresce dall'alto (l'indirizzo più elevato) verso il basso (verso l’indirizzo più basso). L’istruzione "Metti A nello Stack" produce due reazioni:

1. Abbassamento di SP
2. Copiatura di A nel campo di memoria a cui SP Sta puntando in questo momento

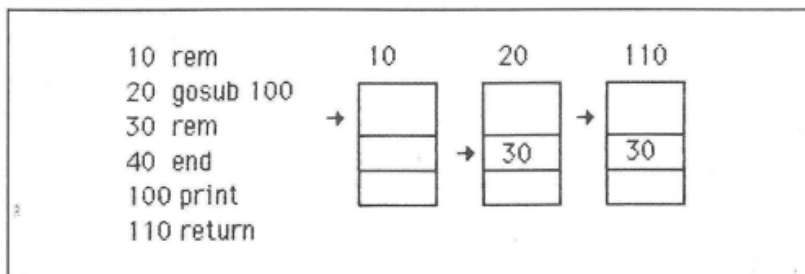



Fig. 2.4: Significato dello Stack per l’esempio in Basic

Il contrario, cioè il prelevamento di A dallo Stack, ha come conseguenza:

1. Copiatura dei dati, ai quali SP sta puntando, su A
2. Aumento di SP

Il significato di tutto ciò è illustrato in Fig. 2.4. Si tratta di sotto programmi, in questo caso, di BASIC. E’ possibile tuttavia immaginare anche i numeri di riga come indirizzi

A destra c'è sempre un pezzo di Stack, e sotto di esso il puntatore. Il comando "GOSUB 100" provoca tre effetti:

1. Abbassamento di SP
2. Immissione del numero di riga successivo (in questo caso 30) nello Stack (nella posizione di memoria alla quale SP sta puntando in questo momento)
3. Salto alla riga 100

Il Return in riga 110 ha come conseguenza:
1. Prelevamento del numero di riga alla quale SP sta puntando:
2. Aumento di SP
3. Salto alla riga 30

Qualcuno si starà certamente chiedendo perché SP viene abbassato da GOSUB e viene di nuovo aumentato da RETURN. Osserviamo quindi la Fig. 2.5. in questo caso il sottoprogramma chiama un secondo sottoprogramma. Ora, dopo la riga 110, ci sono 2 numeri di riga (esattamente indirizzi di RETURN) nella pila. Il RETURN della riga 210 imposta l'SP alla riga 30 e salta quindi a 120, il RETURN della riga 120 riposiziona all'indietro di nuovo SP e salta quindi alla riga 30. L'SP si trova di nuovo al suo valore iniziale, cioè, "siamo tornati a casa".

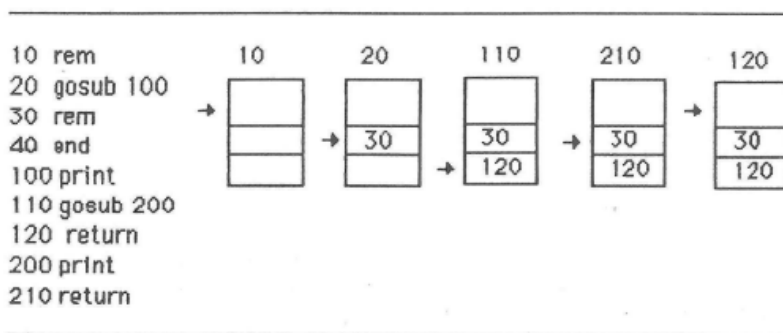



Fig.2.5: Stack nel caso di "Sottoprogramma che chiama un sottoprogramma"

A questo punto è chiaro che con il meccanismo dello Stack è possibile inscatolare a piacere diversi sottoprogrammi. Ogni RETURN aumenta di nuovo l'SP, rispedendolo quindi indietro. Ma facciamo attenzione: cosa succede adesso?

```
10 GOSUB 20
20 GOSUB 10
```

Dal momento che ogni GOSUB abbassa l'SP, ma manca il suo contrario, cioè il RETURN, lo Stack cresce verso il basso. Esso entrerà quindi presto nel nostro codice programma e lo riempirà di indirizzi di RETURN.
Risultato: fallimento totale, anche in BASIC. Provate pure. La seconda applicazione per lo Stack è il trasferimento di parametri ai sottoprogrammi. In linea di principio funziona così: in Assembler (non in BASIC) esistono i comandi "Metti i dati nello Stack" e "Prendi i dati dallo Stack". (Abbiamo già imparato che ogni comando implica una modifica del puntatore dello Stack).
Quindi posso scrivere:

```
10 A sullo Stack
20 B sullo Stack
30 GOSUB 100 (Indirizzo di Return sullo Stack)
```

e quindi nel sottoprogramma:

```
100 Prelevamento indirizzo di Return dallo Stack (e sua annotazione)
110 Prelevamento di B dallo Stack
120 Prelevamento di A dallo Stack
130 Calcolo con A a B
140 Salto all'indirizzo di Return
```

Cosa succede però, se il sottoprogramma deve terminare con RETURN? Scriveremo:

1. Indirizzo di RETURN allo Stack
2. Dati allo Stack
3. GOTO sottoprogramma

Nel sottoprogramma:
1. Dati dallo Stack
2. Lavorare con i dati
3. RETURN

Come già detto, in Assembler non ci sono comandi di PRINT, ma solo la possibilità di scrivere dei Byte in un campo di memoria, che viene rappresentato sul video dal controller video. La programmazione in Assembler significa principalmente spostare dati da un indirizzo ad un altro indirizzo. Anche le apparecchiature periferiche (tastiera, floppy, ecc.) si trovano all'interno del campo indirizzi (detto "memory mapped"). Si accede alle periferiche scrivendo determinati dati in questi indirizzi o leggendoli.

In pratica accederemo raramente all'Hardware, piuttosto immetteremo i parametri in strutture di dati e chiameremo delle routine di sistema, ma anche queste strutture di dati devono venire indirizzate.

Vedete quindi che l’indirizzamento è, di per se stesso, essenziale. Si può accedere ad un indirizzo in tante maniere diverse, come esempio abbiamo già visto l'SP. Posso infatti dire, metti l’SP all'indirizzo 4711. Ma posso anche dire, prendi i dati dall'indirizzo al quale l'SP sta puntando in questo momento (senza sapere a cosa sta puntando).

Già questi sono due modi di indirizzamento. Il 68000 in totale ne conosce 12, di cui ci occuperemo nel prossimo capitolo. Essi sono, per così dire, la chiave per il 68000.

CAPITOLO 3

Indirizzi, dati e comandi

Registri, modi di indirizzamento
Tipi di dati
Struttura dei comandi del 68000

Finora abbiamo imparato che i dati si trovano nelle RAM oppure nelle ROM. Oltre a ciò esiste anche una RAM tutta speciale, che è parte della CPU. Questo campo di memoria è composto di gruppi di 32 Bit, e ciascuno di questi gruppi è chiamato Registro.

3.1 Velocità grazie ai Registri

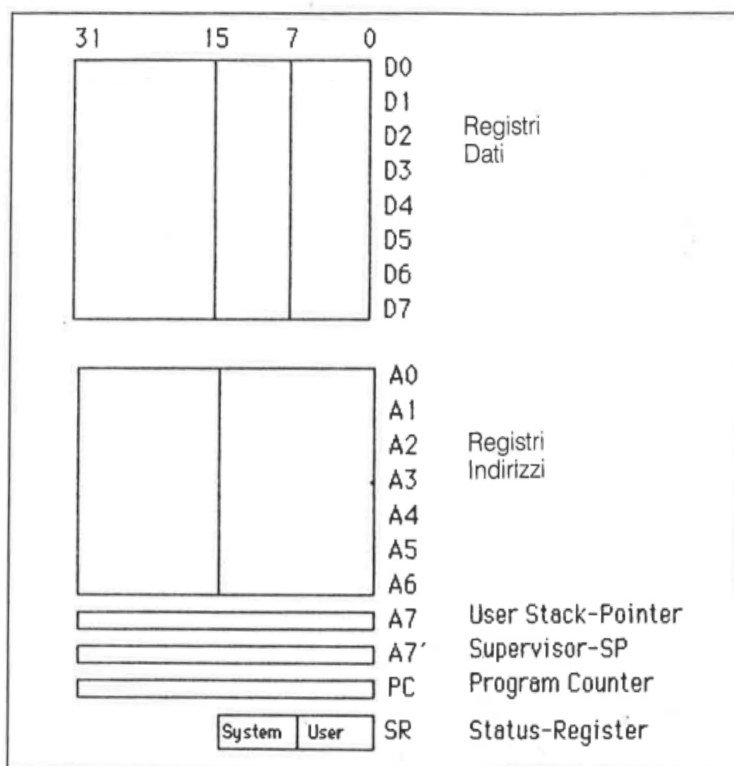



Fig. 3.1: Modello dei registri del 68000

Si accede ai Registri non tramite indirizzi, bensì tramite nomi. Il vantaggio dei Registri, se paragonati al resto della memoria, è che i Registri si trovano sullo stesso Chip

della CPU, e che la CPU supporta l'accesso a questi Registri tramite comandi speciali. Naturalmente non è più necessaria nemmeno la deviazione sulla MMU e sul Bus.
Con ciò, le operazioni dei Registri sono essenzialmente più veloci degli accessi alla memoria centrale ed offrono (grazie ai comandi speciali) un maggiore comfort.

3.2 Il modello Registri del 68000

Da tale punto di vista, una CPU con molti Registri è molto migliore di una CPU con pochi. il 68000 ha molti Registri, cioè:

8 Registri dati, 7 Registri di indirizzo, 2 Puntatori allo Stack, 1 Contatore di programma (PC) e 1 Registro dello stato.

La Fig. 3.1 mostra l'intera gamma di Registri del 68000. Vediamo quindi che i Registri D0-D7 e A0-A7 più PC sono larghi 32 Bit. In figura, i Bit sono numerati da 0 a 31 e danno in tutto 4 Byte.

3.3 Tipi di dati

In un Byte i Bit vengono contati da 0 (Bit più basso) a 7 (Bit più alto). Due Byte (16 Bit) sono chiamati parola. I Bit di una parola vanno da 0 a 15. Due parole (32 Bit) costituiscono una parola lunga. Ecco perché si parla dei tipi di dati Bit, Byte, Parola e Parola lunga. I più grandi numeri rappresentabili,a seconda del tipo, sono mostrati in Fig.3.2.

Bit = 2^1- 1 = 1
Byte = 2^8-1 = 255
Parola = 2^16-1 = 65535
Parola lunga= 2^32-1 = 4.294.967.299

Fig. 3.2: Tipi di dati e valori rappresentabili

A questo punto comprendiamo anche le righe di separazione verticali rappresentate in Fig. 3.1. Nei Registri dati è possibile introdurre dati, parole e parole lunghe. Nei Registri indirizzo il tipo Byte non è possibile. Il puntatore allo Stack (A7) è sempre lungo.

Oltre a ciò, esiste anche il tipo BCD (Binary Coded Decimal). In questo caso un Byte viene suddiviso in due semi-byte (chiamati Nibble). Con i 4 Bit di un Nibble è possibile rappresentare i numeri da 0 a 15. Tuttavia si rinuncia di solito ai numeri da 10 a 15, e restano validi, nella rappresentazione BCD, i valori da 0 a 9. Quindi, in un "BCD Byte" è possibile rappresentare sempre due posizioni decimali. Una parola quindi è sufficiente per un numero decimale a 4 posizioni. Nel caso in cui si abbia bisogno di più posizioni, sarà necessario porre l'uno vicino all'altro la quantità necessaria di Byte. Esistono migliaia di algoritmi complicati relativi al calcolo BCD. Per coloro che non hanno il 68000, tali algoritmi sono molto importanti. Noi invece possiamo rinunciare ad essi, in quanto i comandi adeguati sono già incorporati al 68000.

L'indicazione del tipo è il primo compito

Se si lavora solo con i Registri, la dimensione non ha (quasi) nessuna importanza, quando però si copia un Registro nella RAM, avente tutti i suoi 32 Bit pieni, esso occuperà 4 Byte.
Ciò è molto poco pratico, in quanto spesso sarebbe possibile fare la stessa cosa con molta meno memoria. Di conseguenza nel 68000 esiste la possibilità (diciamo meglio l'obbligo) per ogni operazione che muove dei dati, di indicare di che tipo si tratta.

Un esempio: il trasferimento dati ha luogo solo con il comando MOVE e precisamente con la sintassi "MOVE Sorgente Destinazione". In effetti il comando MOVE non sposta i dati, ma li copia, e copia dalla sorgente alla destinazione.

Al fine di copiare i dati nel Registro D3 all'indirizzo 4711, si scriverà:

```
MOVE.B   D3,4711   oppure
MOVE.W   D3,4711   oppure
MOVE.L   D3,4711
```

Nel caso .B viene copiato un Byte, cioè i Bit da 0 a 7 di D3, nel caso .W viene copiata una parola (Bit 0-15) mentre nel caso .L viene copiato l'intero Registro. Nella memoria centrale (qui a partire dall'indirizzo 4711) vengono posti quindi 1, 2 oppure 4 Byte. In quale sequenza tali tipi di dati si trovino nella RAM, dovremmo già saperlo. Se per es. in D0 c'è la parola $AABB e D0 viene copiato, con un comando MOVE.W, sull'indirizzo 1000, avremo $AA in 1000 e $BB in 1001.
Diciamo ancora una volta chiaramente agli amici che provengono dagli 8 Bit e a quelli che provengono dagli IBM-PC: il 68000 memorizza i dati nella sequenza giusta e non scambia i Byte di una parola, come fanno gli "8 Bit/8088".

A causa di questa ampia gamma di tipi di dati, abbiamo anche l'obbligo di indicare di che tipo si tratta per quasi tutti i comandi. Se manca il tipo, la maggior parte degli Assembler assumono per difetto che si tratti di parola. Torniamo un attimo ai Registri:

la differenza principale fra i Registri dati e i Registri indirizzi è che, nel caso dei Registri indirizzi, i tipi Bit e Byte non sono permessi.

Diversamente sarebbe addirittura possibile memorizzare dei dati nei Registri indirizzi e degli indirizzi nei Registri di dati. Quest'ultima cosa la incontreremo spesso nell'Amiga, dal momento che molte routine si aspettano anche degli indirizzi nei Registri dati.

Il Registro dello stato ha uno scopo molto particolare. Con esso viene realizzato in Assembler l'IF-THEN; ne riparleremo in seguito.

3.4 Comandi

Quanti comandi esistano in Assembler, non è facile a dirsi, Ciò dipende dal fatto che un comando produce effetti completamente diversi tra loro, a seconda del tipo di indirizzamento e di altre varianti. Cominciamo con la struttura del comando. Un comando può avere: nessun operando, oppure uno, oppure due. Gli operandi possono essere contenuti già nella parola comando oppure possono occupare fino ad un massimo di 4 parole, che seguiranno immediatamente la parola comando. Non preoccupiamoci per il momento di ciò. Nel testo sorgente si scriverà semplicemente il comando e gli operandi, sarà l'Assemblatore ad occuparsi di quante parole essi costituiscono.

Un esempio per un comando senza operandi è RTS (Return from Subroutine) che corrisponde al RETURN in BASIC. Il comando "CLR D0" ha un operando. Esso significa Clear (Cancella (Riempi con Bit 0)) l'operando D0 (il Registro D0). Un esempio per un comando con due operandi è:

```
MOVE.L A3,A4
```

in questo caso la parola lunga del Registro A3 viene copiata su A4.

3.5 Significato e scopo dei tipi di indirizzamento

La duttilità di una CPU dipende dal numero di tipi di indirizzamento significativi che essa può gestire, ed in questo caso il 68000 brilla di luce tutta propria. Tutto questo lusso, tuttavia, rende la questione ancora più complicata, in quanto tutto ciò ha bisogno di venire appreso. Ci troviamo, inoltre, di fronte ad una barriera, per gli amici che sono abituati a lavorare con i pochi (e primitivi) tipi di indirizzamento degli "8 Bit". D'altra parte, quando si sarà padroni di questo argomento, si sarà padroni anche del 68000.

Vediamo meglio di cosa si tratta: nell'esempio di cui sopra

```
MOVE.L A3,A4
```

Il contenuto del Registro A3 è stato copiato su A4. Se io al contrario scrivo:

```
MOVE.L (A3),(A4)
```

ciò significa che i contenuti dei Registri devono venire visti come indirizzi. Se per es., al momento del comando, A3 ha il valore 4711 ed A4 è 5711, verrà copiata una parola lunga dall'indirizzo 4711 (partendo da esso e Byte per Byte) sull'indirizzo 5711.

Ora abbiamo imparato a conoscere due tipi di indirizzamento, cioè il "Registro diretto" (MOVE.L A3,A4) ed il "Registro indiretto" (MOVE.L (A3),(A4)).

Per salire ancora di un passo, osserviamo il "Registro indirizzi indiretto con post-incremento". Vedremo quanto segue:

```
MOVE.W (A0)+,D0
```

Detto chiaramente: copia la parola, sulla quale A0 sta puntando, su D0 ed aumenta dopo ciò (post) A0 di 2. Due è spiegato dal fatto che una parola ha 2 Byte. E' importante notare che il 68000 e una macchina-Byte, e che ogni indirizzo punta ad un Byte.
"MOVE.(A0)+ ,D0" farebbe copiare una parola lunga e dopo ciò farebbe incrementare A0 di 4. La variante successiva sarebbe "Registro indirizzi indiretto con pre-decremento". Un esempio:

```
MOVE.L D0,-(A5)
```

In questo caso il 4 viene prima di tutto (pre) sottratto da A5 (una parola lunga ha 4 Byte), quindi D0 viene copiato nel punto al quale A5 sta puntando. Abbiamo già visto qualcosa di simile parlando dello Stack nel Capitolo 2. I dati vengono portati nello Stack, mentre si abbassa il puntatore allo Stack (SP), quindi i dati vengono copiati sull'indirizzo al quale SP sta puntando.

I dati vengono prelevati dallo Stack, prendendoli dall'indirizzo al quale SP sta puntando, quindi SP viene aumentato. Ciò sarebbe di nuovo il nostro già noto

```
MOVE.L (A5)+,D0
```

Effettivamente ogni Registro indirizzi può venire utilizzato come puntatore allo Stack.

La particolarità del Registro A7, che in molti Assembler si chiama anche SP, sta nel fatto che si può accedere a questo Registro anche tramite comandi come JSR (Jump to

Subroutine) e RTS (Return). Ciò, in verità, può venire eseguito anche con altri Registri, per es. invece di RTS si può anche scrivere:

```
MOVE.L   (A7)+,A0
JMP      (A0)
```

Il comando MOVE prende l'indirizzo Return dallo Stack e lo mette nel Registro A0, quindi ha luogo un salto (Jump) all'indirizzo al quale A0 sta puntando. Il lettore si chiederà perché tutto ciò, un RTS è molto più semplice. Ha ragione, ma vedrà meglio questa soluzione nei programmi che per es. devono venire chiamati dal BASIC, e cioè:

```
MOVE.L (A7)+, 4711
```

molti altri comandi

```
MOVE.L   4711,A0
JMP      (A0)
```

Con il primo comando MOVE, l'indirizzo Return viene prelevato da una posizione sicura nella RAM. Si dice anche che l'indirizzo di Return viene salvato. Se qualcosa va storto, con questo indirizzo posso sempre tornare al BASIC, indipendentemente da dove si trova in quel momento il puntatore allo Stack. Dopo questa divagazione nella pratica, che doveva comunque spiegare per quale motivo si ha bisogno di tipi di indirizzamento diversi fra loro, torniamo alla teoria. La Fig. 3.3 mostra una lista di tutti i tipi di indirizzamento.

Notiamo immediatamente che un indirizzo può essere composto da diverse indicazioni. Da esse la CPU calcola l'indirizzo definitivo, detto anche indirizzo effettivo (ea).
Come già sappiamo, la parola comando occupa 16 Bit. I 4 Bit più alti descrivono il comando in sé e per sé. I rimanenti 12 sono suddivisi in due gruppi da 6 Bit, che indicano il tipo di indirizzamento di destinazione e sorgente (se disponibile). Anche i 6 Bit per operando si suddividono in due gruppi da 3 Bit, un gruppo si chiama Modo, il secondo Registro. Con 3 Bit sono rappresentabili i numeri da 0 a 7, di conseguenza esiste anche il Registro da A0 a A7 oppure da D0 a D7. Esistono tuttavia più di sette
tipi di indirizzamento, con il risultato che non tutti i Registri sono permessi per ogni tipo di indirizzamento. La cosa più importante da sapere è che determinati tipi di indirizzamento non sono permessi per gli operandi di sorgente e contemporaneamente per gli operandi di destinazione, ed inoltre possono non essere permessi anche per qualche comando. A questo proposito troveremo ulteriori informazioni nell'Appendice. In Fig.3.3 sono rappresentati sia il Modo che il Registro sotto forma di numeri binari. Quando c'è “An” oppure “Dn”, sarà possibile inserire al loro posto %000 fino a %111 (in decimale, da 0 a 7).

Tipo di indirizzamento	Abbreviazione	Modo	Registro
Registro dati diretto	Dn	000	Dn
Registro indirizzi diretto	An	001	An
Reg. indirizzi indiretto (ARI)	(An)	010	An
ARI con post-incremento	(An)+	011	An
ARI con pre-decremento	-(An)	100	An
ARI con distanza di indirizzamento	d16(An)	101	An
c.s. + Indice	d8(An,Rn)	110	An
Corto assoluto	$XXXX	111	000
Lungo assoluto	$XXXXXXXX	111	001
Relativo al PC con distanza di indirizzamento	d16(PC)	111	010
Relativo al PC con distanza di indirizzamento + Indice	d8(PC,Rn)	111	011
Costante, Registro dello stato	#,SR,CCR	111	100

Fig. 3.3: Elenco di tutti i tipi di indirizzamento

3.6 Tipi di indirizzamento in dettaglio

A questo punto dovremmo aver capito che dalla parola comando e dai tipi di indirizzamento in essa codificati deriva anche il numero di parole che il comando occuperà nella memoria. L'indirizzamento da Registro a Registro (per es. MOVE A0,A1) produce una parola, mentre se si dà un indirizzo assoluto, se ne otterranno almeno due. Questo significa che nei 16 Bit della parola comando sono contenute tutte le informazioni necessarie alla CPU al fine di decodificare il comando stesso.

E’ in questo modo che funzionano anche i cosiddetti disassemblatori, che sono dei programmi che formano dal codice in linguaggio macchina il testo in chiaro del linguaggio Assembler. Se tuttavia il lettore non vuole scrivere un programma di questo genere, questi campioni di Bit saranno per lui assolutamente indifferenti. Esistono tuttavia molti Hacker (quel dilettanti che per hobby tentano di accedere ai sistemi di grossi Enti o Società, infrangendone i codici) che sono in grado di leggere il codice
esadecimale (il linguaggio macchina) degli “8Bit” con la stessa semplicità con cui la gente comune legge il giornale. Questa capacita è comunque inutile per il 68000, quindi procediamo ad analizzare la questione dal punto di vista pratico, cioè tutti i tipi di indirizzamento con un esempio ciascuno.

3.6.1 Registro diretto

Si accede direttamente ad uno dei Registri, Esempio:

```
CLR D0 (cancella D0)
```

3.6.2 Registro di indirizzamento indiretto (ARI)

Il contenuto del Registro è un indirizzo, sul quale l'operazione avrà il suo effetto.
Esempio:

```
MOVE(A0),D0  Notare l'ARI!
```

La parola il cui indirizzo si trova in A0, viene copiata su D0.

3.6.3 ARI con post-incremento

Funziona come l'ARI, solo che alla fine il Registro viene incrementato.
Esempio:

```
MOVE.B   (A0)+,D0 ;Copia, quindi A0=A0+1
MOVE.W   (A0)+,D0 ;Copia, quindi A0=A0+2
MOVE.L   (A0)+,D0 ;Copia, quindi A0=A0+4
```

3.6.4 ARI con pre-decremento

Come sopra, solo che il Registro prima dell'operazione viene abbassato. Esempio:

```
MOVE -(A0),D0    ;A0=A0-2, quindi copia
```

3.6.5 ARI con distanza di indirizzamento

L'indirizzo effettivo è la somma del contenuto del Registro più la distanza di indirizzamento. La distanza di indirizzamento è un numero a 16 Bit preceduto da un segno nel campo -32668...32767.
Esempio:

```
MOVE -100(A0),D0
```

Se A0 fosse uguale a 500, la parola verrebbe copiata dall'indirizzo 400 a D0. Questo tipo di indirizzamento dovrà venire ben memorizzato, in quanto con l'Amiga ne avremo bisogno spesso.

3.6.5.1 ARI con distanza di indirizzamento e Indice

A questo punto la cosa si fa più complessa. La distanza di indirizzamento è ora un numero a 8 Bit preceduto da un segno nel campo da -128...127. A questo punto può venire indicato solo un altro Registro. Esempio:

```
MOVE 100(A0,D0),4711
```

100 è la distanza di indirizzamento, A0 contiene l'indirizzo di base, in D0 è contenuto l'indice. Tutti e tre vengono addizionati. La somma è un indirizzo, e la parola (Byte, Parola lunga) che in esso si trova, viene copiata nella destinazione (in questo caso indirizzo 4711).

Per l'Indice può venire indicato anche un altro tipo, cioè sarebbero permessi anche D0.B oppure D0.L, mentre in caso di Registro di indirizzamento come Indice sono permessi naturalmente solo An.W e An.L (0< n< 7). Anche l’Indice è preceduto da un segno, affinché, anche in caso di parola lunga, rientri nel campo di 2 Giga-Byte. Questo comando è ideale per l’elaborazione di tabelle e di matrici. Spesso la distanza di indirizzamento non è necessaria (la variabile si trova nel Registro Indice), questo è il motivo per cui spesso è possibile incontrare per es. “0(A3,D4.L)”.

3.6.6 Indirizzamento assoluto

Si tratta del caso più semplice. Esempio:

```
MOVE 4711,5713
```

La parola dall'indirizzo 4711/12 viene copiata sull'indirizzo 5713/14. La CPU si occupa solo della differenza fra corto e lungo (campo di indirizzamento solo 64 Kbyte oppure tutti i 16 Mbyte del 68000). L'utente noterà appena la differenza (l’indirizzo lungo ha bisogno di più Byte nella lunghezza del comando ed è un po' più lento).

3.6.7 Indirizzamento costante

Anche questo è molto semplice. Al fine di muovere una costante, è necessario farla precedere solo dal segno di #. Per caricare per es. il carattere ASCII A nel Registro D0, scriveremo:

```
        MOVE #65,D0
oppure  MOVE #’A’,D0
```

3.6.8 Indirizzamento relativo al PC

Per poter sfruttare questa caratteristica, è necessario fare alcune premesse. Se in un programma in Assembler viene indicato un indirizzo assoluto, il programma è legato ad un determinato punto della memoria. Per es. anche “(A0)” (indiretto) è assoluto in questo senso, in quanto si sarebbe dovuto procurare anche ad A0 un indirizzo.

Il campo di indirizzi programmato dall'utente può tuttavia essere già completo, per cui il programma dovrà poter accedere anche ad altri punti. Per fare ciò abbiamo due possibilità. Una è che il programma sia spostabile (rilocabile). E' l'Assembler che si

occupa di ciò, quando memorizza insieme con il programma una tabella di tutti gli indirizzi assoluti. Il caricatore (oppure il programma stesso oppure una Utility può correggere questi indirizzi quando questi sommano la differenza fra l'indirizzo di partenza programmato e quello effettivo a tutti gli indirizzi assoluti come da tabella. La seconda possibilità è quella di scrivere il programma indipendente dalla posizione (Position Independent). In un programma di questo genere non potranno esserci indirizzi assoluti, e ciò permette al 68000 di effettuare l'indirizzamento relativo al PC. L'indirizzo viene calcolato come posizione attuale del PC più Offset. Anche in questo caso l'Offset è limitato a -32768...32767.
Esempio:

```
MOVE 100(PC),D0
```

Relative al PC con distanza di indirizzamento e Indice

Qui vale quanto detto per "ARI con distanza di indirizzamento e Indice", solo che l'indirizzo di base in questo caso è PC + 2. Esempio:

```
MOVE 100(PC,A0.W),D0
```

Per quanto concerne la teoria, avremmo finito. Ci mancano ancora molte cose, ma esse verranno spiegate nei punti adeguati sulla base di esempi pratici. Nel prossimo capitolo cominciano i primi listati. Dovremo inoltre preoccuparci del funzionamento dell'Editor, dell'Assemblatore, del Linker, e del Processore Batch.

Relativamente al DOS dovremmo ancora aggiungere qualcosa, tutta via non vogliamo reinventare la ruota, ma semplicemente sfruttare al meglio ciò che in esso è contenuto.

CAPITOLO 4

Entriamo nella pratica

Exec e DOS
Funzionamento dell'Assemblatore
l primi listati

4.1 Corso veloce sul DOS

Il sistema operativo dell'Amiga è costituito, se semplifichiamo grossolanamente, da tre parti, cioè:

DOS
Intuition
Exec

Il compito di ogni OS (Operating System = Sistema Operativo) è quello di creare il collegamento fra il computer e il mondo esterno. Compiti quali la lettura di caratteri da tastiera, la rappresentazione di caratteri sul monitor oppure la lettura di dati da un dischetto, sono funzioni tipiche di un OS.

Ovviamente l'Amiga può venire fatto funzionare come un computer standard (cosa che accade quando si è in CLI) oppure tramite l'area utente grafica chiamata “Workbench”.
Sempre molto grossolanamente, si potrà quindi dire (approfondiremo meglio in seguito):

Standard = DOS
Grafica = Intuition

Resta quindi Exec, il quale, nel nostro modello semplificato, si occupa principalmente del Multitasking.

Il DOS (Disk Operating System = Sistema Operativo su Disco) ha a che vedere anche con i dischetti, tuttavia il nome non gli è degno. In effetti il DOS può occuparsi anche della tastiera e del monitor, quindi aprire le tipiche finestre Amiga, gestire la stampante e altro. Dal momento che il DOS è molto facile da gestire, ci occuperemo dapprima di esso. Quindi sarà necessario imparare dapprima la programmazione in Assembler in sé e per sé, e non è poco. Tralasciamo per il momento le parti complicate del Software di sistema dell'Amiga.

Dal punto di vista del programmatore, il DOS è un insieme di routine (sottoprogrammi) tutte utilizzabili. Alcune di esse, come per es. il caricamento e l'avviamento di un programma applicativo, sono accessibili all'utente normale, mentre tutte lo sono al programmatore in Assembler.

Ciascuno di questi sotto programmi comincia naturalmente ad un determinato indirizzo, e di conseguenza sarà possibile chiamare un programma di questo genere con la forma “JSR Indirizzo”. Ciò in pratica non viene effettuato, in quanto ogni modifica nell'OS condurrebbe allo spostamento di qualche indirizzo o di tutti, per cui tutti i programmi “vecchi” diverrebbero solo carta straccia.

4.2 Chiamata di routine DOS

Al fine di essere indipendenti dagli indirizzi assoluti, la maggior parte degli OS funziona secondo il seguente schema.

Tutti i sottoprogrammi ricevono un numero, chiamato numero di funzione. L'OS contiene una tabella nella quale viene annotato quale indirizzo appartiene ad ogni numero di funzione. L'OS ha quindi una routine, il cui indirizzo non cambia mai. Essa viene chiamata routine di smistamento. Per poter chiamare un sotto programma, si fornisce alla routine di smistamento il numero di funzione. Essa quindi calcolerà (tramite la tabella) l'indirizzo della routine in questione e la chiamerà.

L'Amiga fa la stessa cosa ma in maniera più raffinata e quindi più futuribile. Lo svantaggio del metodo standard è infatti la difficoltà di aggiungere nuove routine (la tabella si trova nella ROM). Nell'Amiga invece le tabelle si trovano nella ROM oppure nella RAM oppure sul dischetto. Tali tabelle costituiscono una parte delle cosiddette Libraries (Biblioteche).

Libraries: chiavi per l'Amiga

Una Library, semplificando, è una raccolta di sottoprogrammi con una tabella ad essa attribuita (un input per ogni sottoprogramma). Esiste una Library per ogni scopo (es. DOS, Intuition, Grafica). Se si vuole utilizzare una funzione di una Library, sarà necessario aprire la Library con "OpenLibrary". Questa funzione fornisce un puntatore all'inizio (Indirizzo di partenza) della tabella. Al fine di poter chiamare un sottoprogramma, sarà necessario indicare l'indirizzo di partenza della tabella ed un cosiddetto Offset, che è la differenza fra l'indirizzo di partenza e la relativa posizione di tabella.

Tutto ciò viene gestito dal cosiddetto "Library-Manager" (una parte di Exec). Il Manager sa se una Library si trova nella ROM o nella RAM. Diversamente, cercherà di caricare la Library dal dischetto. Se anche questo non funziona (la Library non è sul dischetto oppure la memoria è già piena) risponde con 0 come indirizzo.

Tutto ciò ha anche un altro motivo: possediamo un Amiga che si differenzia dai suoi concorrenti anche grazie al suo sistema Multitasking. Ciò significa, semplificando, che diversi Task (Programmi) possono utilizzare una Library quasi contemporaneamente.
Il primo Task caricherà se necessario la Library dal dischetto nella RAM (più precisamente ne provocherà il caricamento). Se altri Task aprono la stessa Library, il Manager fornirà loro solo l'indirizzo. Da ciò deriva che una Library può venire cancellata dalla memoria solo quando l'ultimo Task ha detto di non averne più bisogno.

Per fare ciò esiste la funzione “CloseLibrary”. Ogni Task (quindi ogni programma scritto dal lettore) dovrà chiudere tutte le Library che ha aperto. Diversamente la memoria potrebbe non bastare.

4.3 Costituzione di un programma in Assembler

Ogni programma in Assembler è composto dai campi Marcatura. Comandi, Operandi (se presenti) e Commento. Ecco un esempio:

Marcatura	Comando	Operando (i)	Commento

```
Start           clr             d0              ;Cancella Registro
                Move            d0,d1           ;Comando con 2 operandi
weiter                                          ;Marcatura nella riga
                rts                             ;Nessun operando
```

Il Commento non è obbligatorio, ma contribuisce molto alla leggibilità. A seconda del tipo di Assembler, esso dovrà cominciare con un punto e virgola oppure con un asterisco, mentre per altri Assembler è sufficiente la sua posizione per identificarlo (Campo Commento). Se il Commento si trova da solo in una riga, esso dovrà necessariamente preceduto da “;” oppure “*”.

La Marcatura (Label) viene utilizzata solo in alcuni casi. Essa potrà anche trovarsi da sola in una riga, ma avrà effetto sempre solo a partire dalla riga successiva contenente un comando. In alcuni Assembler la Marcatura deve seguire un due-punti, ma solo quando si trova in un Campo Marcatura e non quando viene citata.

Ad esclusione dei casi particolari “solo Marcatura” oppure “solo Commento”, un comando deve trovarsi in una riga insieme con i suoi Operandi (se presenti). I singoli campi devono venire separati l'uno dall'altro per lo meno da uno spazio bianco. Per la separazione si utilizza principalmente il tasto di tabulazione (distanza di 8 spazi). In questo caso sarà comunque necessario usare un Editor di testi che produca degli spazi bianchi quando si preme il tasto di tabulazione.

4.4 Il primo listato: Output di una stringa

in Fig. 4.1a è contenuto il nostro primo programma:

```
* A1_Met   Il mio primo Programma   !!! Versione Metacomco  !!!
* -------------------------------------------------------------

        INCLUDE "libraries/dos_lib.i"
        XREF    _DOSBase
        XREF    _SysBase
        XREF    _LVOOpenLibrary
        XREF    _LVOCloseLibrary
        XDEF    _main

_main    move.l   #dosname,a1             ;Nome della DOS-Lib
         moveq    #0,d0                   ;Versione indifferente
         move.l  _SysBase,a6              ;Base di Exec
         jsr      LVOOpenLibrary(a6)      ;Apertura della DOS-Lib
         tst.l    d0                      ;Errore?
         beq      fini                    ;Se errore, Fine
         move.l   d0,_DOSBase             ;Annotare il puntatore

* Determinazione dell'Handle di output:

         move.l   _DOSBase,a6             ;Chiamata funzione DOS
         jsr      _LVOOutput(a6)          ;Prelevamento Handle di Output
         move.l   d0,d4                   ;e sua annotazione in d4

* ora Output di testo:

         move.l   d4,d1                   ;Handle di Output
         move.l   #string,d2              ;Indirizzo del Testo
         moveq    #20,d3                  ;Lunghezza del Testo
         move.l   _DOSBase,a6             ;Base del DOS
         jsr      _LVOWrite(a6)           ;Funzione "Scrittura"

* Al termine chiudere sempre le Lib!

         move.l   _DOSBase,a1             ;Base delle Lib
         move.l   _SysBase,a6             ;Base di  Exec
         jsr      _LVOCloseLibrary(a6)    ;Funzione "Chiusura"

fini     rts                              ;Ritorno al CLI

* campo dati:


dosname  dc.b     'dos.library',0
                  cnop     0,2

string            dc.b     'Ciao, caro lettore!!',10
                  cnop     0,2

                  end
```

Fig 4.1a: Output di una stringa (Assemblatore Metacomco)

Come introduzione presentiamo il listato tre volte, cioè una volta per l'Assembler Metacomco, una per quello SEKA e una per quello DEVPAC. Tutti gli altri listali valgono per l'Assembler DEVPAC della HiSoft. I lettori che possiedono altri Assembler dovranno essere in grado, sulla base delle indicazioni fornite nel presente testo, di adattare i listati. Le tabelle LVO contenute nell'Appendice sono destinate in particolare agli utilizzatori del SEKA. Cominciamo con il Metacomco, che funziona secondo il metodo più classico.

Il programma dovrà scrivere sul video "Ciao, caro lettore!!" e quindi tornare al CLI.

Tralasciamo per il momento l'Overhead e osserviamo il listato a partire dalla riga che comincia con "_ main".

Anche per il più semplice dei programmi dobbiamo aprire una Library. Per fare ciò abbiamo bisogno della funzione OpenLibrary, che si trova nella Exec-Library. Poi, per ogni funzione, abbiamo sempre bisogno dell'indirizzo di base della Library, ottenibile anch'esso con OpenLibrary. Al fine di evitare che il gatto si morda la coda, nell'Amiga c'è un indirizzo fisso (l'unico) che è la base di Exec. Questo indirizzo (4) ha il nome simbolico _AbsExecBase oppure (in uso) _SysBase.

La chiave nascosta si trova nelle righe seguenti:

```
        move.l  _SysBase,a6             ;Base di Exec
        jsr      LVOOpenLibrary(a6)     ;Apertura della DOS-Lib
```

In chiaro: carica il Registro a6 con la costante _SysBase. Quindi salta al sottoprogramma (JSR = Jump to Subroutine), il cui indirizzo viene calcolato dalla costante _LVOOpenLibrary e dal Registro a6 (tipo di indirizzamento ARI con Offset,Ved. Cap. 3).

Prima di procedere dobbiamo anche dire quale Library deve venire aperta. Per fare ciò è necessario fornire due parametri, cioè il nome della Library e il numero di versione. Ciò viene eseguito dalle righe

```
    _main       move.l  #dosname,a1             ;Nome della DOS-Lib
                moveq   #0,d0                   ;Versione indifferente
```

La prima riga significa: copia (move) indirizzo di dosname nel Registro a1. Il simbolo di # (cancelletto) è importantissimo. Esso significa in questo caso "indirizzo di". Se tale simbolo viene dimenticato, si otterrà una interruzione immediata, in quanto il significato sarebbe "contenuto di". A questo punto resta ancora il numero della versione: possono esistere delle Libraries che si differenziano l'una dall'altra non nel nome bensì nel numero di versione. La versione 0 non potrà mai esistere.

0 è riservato solo per “prendi la prima versione” (di solito l'unica disponibile). Dopo il JSR, il sotto programma ritorna indietro, e nel Registro d0 ci sarà ora l'indirizzo di base della Library DOS. Questo indirizzo viene messo al sicuro immediatamente nella variabile _DOSBase.

A questo punto ci siamo procurati l'indirizzo della DOS-Library e possiamo lavorare con essa. Al fine di far uscire un testo, dobbiamo dapprima sapere dove il testo dovrà venire scritto. Può trattarsi per es. di un file. Dal punto di vista del DOS, tutta via anche l'apparecchiatura di Output corrente costituisce un file con il nome specifico Output. All'avviamento dell'Amiga (e fino a che non verrà modificato) per Output si intende il monitor (detto più esattamente: la finestra CLI). Per il DOS un File oppure l'accesso a File ha luogo tramite i cosiddetti Handle (routine per la gestione di una unità esterna).
Normalmente un file viene aperto con Open. Ma il nostro file di Output è già aperto, e di conseguenza abbiamo una funzione con il nome _LVOOutput, che determina gli Handle dell'Output. Ciò accade con le seguenti righe:

```
move.l   _DOSBase,a6      ;Chiamata Funzione DOS
jsr      _LVOOutput(a6)   ;Prelevamento Handle di Output
move.l   d0,d4            ;e sua annotazione in d4
```

Le funzioni DOS vengono chiamate in linea di massima come le funzioni Exec. La differenza sta solo nel fatto che ora il Registro a6 punta alla base della DOS-Library (_DOSBase). Anche la costante _LVOOutput è definita in maniera diversa (che vedremo in seguito). Come tutte le funzioni, anche _LVOOutput mette il risultato in d0.

Dal momento che d0 è un Registro, che viene utilizzato anche da altre funzioni, salviamolo (copiamolo) nel Registro d4, in ogni caso, ci troviamo ad avere ora l'Handle nel Registro d4 e possiamo quindi lavorare.

Al fine di scrivere in un file (oppure in una apparecchiatura) il DOS ha bisogno dei seguenti parametri:
- Handle in d1
- Indirizzo a partire dal quale si trovano i dati, in d2
- Numero dei Byte di dati in d3

Osserviamo ora queste righe, e ritroveremo:

```
move.l   d4,d1            ;Handle di Output.
move.l   #string,d2       ;Indirizzo del Testo
moveq    #20,d3           ;Lunghezza del lesto
```

Dopo averla preparata in questo modo, possiamo chiamare la funzione _LVOWrite:

```
move.l _DOSBase,a6        :base di DOS
jsr _LVOWrite(a6);Funzione “Scrittura”
```

E' chiaro che il principio è sempre lo stesso:

```
move.l   Indirizzo_Base,a6
jsr      Offset(a6)
```

Alla fine del programma, la DOS-Library verrà richiusa con lo stesso metodo.

Nel campo dati troviamo ora alcune istruzioni in Assembler. L'istruzione in Assembler “dc.b” è importante. Prestiamo quindi attenzione: si tratta di una istruzione per L'Assemblatore, non di un comando per il 68000. “dc” significa “define constant” (definisci costante), “dc.b” significa quindi una costante del tipo Byte.

```
dosname  dc.b     'dos.library',0
         cnop     0,2

string   dc.b     'Ciao, caro lettore!',10
         cnop     0,2
```

“string” è una Label, e tutta l’istruzione all'Assemblatore suona cosi: imposta a partire dall'indirizzo (simbolico) string la sequenza di caratteri “ciao...”. Infatti vogliamo fare apparire “ciao”. Per fare ciò abbiamo bisogno di un puntatore, che punti a “ciao” (più precisamente: dapprima alla c). Per ottenere ciò, chiamiamo il Registro d2. Affinché d2 venga caricato con l'indirizzo di string, scriviamo la riga

```
move.l   #string,d2       ;Indirizzo del Testo
```

Ripetiamo: “dc.b” significa “definisci costante” e nel nostro caso del tipo Byte. Nel campo degli Operandi si trovano quindi i Byte. Questi ultimi possono venire immessi singolarmente (dc.b 100,33,20) come testo fra virgolette, oppure, come nel nostro esempio, tutti insieme. Il nome di una Library deve venire concluso con un Byte 0, ecco perché abbiamo uno zero alla fine del primo "dc.b".

Per la seconda stringa non c'è bisogno del Byte di zero, in quanto la funzione Write si aspetta la lunghezza come parametro. Il 10 alla fine di questo testo e il codice ASCII per “nuova riga”. Con ciò faccio solo in modo che dopo il funzionamento del programma, il Prompt-CLI (1>) inizi su una nuova riga.

“cnop 0,2” è un'altra forma per “pari” che ho scelto perché tale forma viene capita contemporaneamente sia dall'Assembler della HiSoft che da quello della Metacomco (nel SEKA esso è invece ALIGN). Ricordiamo quindi: si dovrebbe far cominciare un testo sempre da un limite di parola. Cosa ancora più sicura (e obbligatoria per certe funzioni) è un limite di parola lunga (cnop 0,4).

Adesso resterebbe ancora una domanda da chiarire, e cioè da dove vengono le costanti come per es. _LVOOpenLibrary. Osserviamo il listato SEKA in Fig. 4.1.b.

```
* A1_Seka  Il Mio Primo Programma !!! Seka-Version !!!
* ----------------------------------------------------

SysBase: equ      4                 ;Base di Exec
LVOOpenLibrary:   equ      -552             ;Apertura Library
LVOCloseLibrary:  equ      -414             ;Chiusura Library

LVOOutput:        equ      -60              ;DOS: prelevamento Output Handle
LVOWrite:         equ      -48              ;Output

*Apertura di DOS/Lib:

main:    move.l   #dosname,a1               ;Nome della DOS-Lib
         moveq    #0,d0                     ;Versione indifferente
         move.l   SysBase,a6                ;Base di Exec
         jsr      LVOOpenLibrary(a6)        ;Apertura di DOS-Lib
         tst.l    d0                        ;Errore?
         beq      fini                      ;se Errore, Fine
         move.l   d0,DOSBase                ;Annotare il puntatore

* Determinazione dell'Handle di Output:

         move.l   DOSBase,a6                ;Chiamata Funzione DOS
         jsr      LVOOutput(a6)             ;Prelevamento Handle di Output
         move.l   d0,d4                     ;e sua annotazione in d4

* ora Output di testo:

         move.l   d4,d1                     ; Handle di Output
         move.l   #string,d2                ;Indirizzo del Testo
         moveq    #20,d3                    ;Lunghezza del Testo
         move.l  DOSBase,a6                 ;Base del DOS
         jsr      LVOWrite(a6)              ;Funzione "Scrittura"

* Al termine chiudere sempre le Lib!

         move.l   DOSBase,a1                ;Base della Lib
         move.l   SysBase,a6                ;Base di  Exec
         jsr      LVOCloseLibrary(a6)       ;Funzione "Chiusura"

fini:    rts                                ;Ritorno al CLI

* Campo dati:

DOSBase: dc.l     0
         align    4
dosname: dc       'dos.library',0
          align   4

string:  dc       'Ciao, caro lettore!!',10
         align    4
```

Fig 4.1b: Stesso programma in versione SEKA

Nelle prima righe troviamo i cosiddetti Equates (uguaglianze). Anche in questo caso si tratta di istruzioni per l'Assembler.

```
        SysBase: equ      4
```

Definisce la costante SysBase con valore 4. Secondo questo concetto, i seguenti modi di scrittura sono uguali:

```
        move.l SysBase,a6
        move.l. 4,a6
```

In linea di massima si dovrebbe scegliere comunque la prima forma. Sia nell'Assembler Metacomco che in quello DEVPAC ci sono dei file Include (moduli di testo) nei quali queste costanti sono definite. Inoltre c'è da notare che tutti gli Offset cominciano con _LVO. LVO significa Library Vector Offset. Nel caso del SEKA, oltre a DOS.Lib (che è solo 1/16 di tutto) non esiste nient'altro. Le differenze del SEKA rispetto allo standard possono venire mostrate meglio sulla base della soluzione DEVPAC di cui alla Fig. 4.1.c.

```
        opt      l-                       ;non linkare!


* A1_Dev Il Mio Primo Programma !!! DevPac-Version !!!
* ------------------------------------------------------

_SysBase   equ    4                       ;Base di Exec
_LVOOpenLibrary  equ      -552            ;Apertura Library
_LVOCloseLibrary equ      -414            ;Chiusura Library


_LVOOutput        equ     -60             ;DOS: prelevamento Output Handle

_LVOWrite         equ     -48             ;Output

*Apertura di DOS/Lib:

_main   move.l   #dosname,a1              ;Nome della DOS-Lib
        moveq    #0,d0                    ;Versione indifferente
        move.l  _SysBase,a6               ;Base di Exec

        jsr      _LVOOpenLibrary(a6)      ;Apertura di DOS-Lib
        tst.l    d0                       ;Errore?
        beq      fini                     ;se Errore, Fine
        move.l   d0,DOSBase               ;Annotare il puntatore

* Determinazione dell'Handle di Output:

        move.l   _DOSBase,a6              ;Chiamata Funzione DOS
        jsr      _LVOOutput(a6)           ;Prelevamento Handle di Output
        move.l   d0,d4                    ;e sua annotazione in d4
```

```
* ora Output di testo:

        move.l   d4,d1                   ;Handle di Output
        move.l   #string,d2              ;Indirizzo del Testo
        moveq    #20,d3                  ;Lunghezza del Testo
        move.l   DOSBase,a6              ;Base del DOS
        jsr      _LVOWrite(a6)           ;Funzione  "Scrittura"

* Al termine chiudere sempre le Lib!

        move.l   _DOSBase,a1             ;Base di Lib
        move.l   _SysBase,a6             ;Base di Exec
        jsr      _LVOCloseLibrary(a6)    ;Funzione "chiusura"

fini    rts                              ;Ritorno al CLI

* Campo dati:

_DOSBase dc.l    0

dosname dc.b     'dos.library',0
        cnop     0,2

string  dc.b     'Ciao, caro lettore!!',10
        cnop     0,2
```

Fig. 4.1.c: Stesso programma in versione DEVPAC

Nel Metacomco e nel DEVPAC, tutte le costanti cominciano con il trattino di sottolineatura. Nel caso del SEKA tale trattino di sottolineatura non è permesso come primo carattere. “dc.b” non è noto al SEKA, bisogna tralasciare il b.

Per DOS_Base ho occupato questo indirizzo con la parola lunga 0, tramite “dc.l 0”.
Per questa variabile avrei bisogno di spazio in memoria, per cui di solito si scrive normalmente “ds.l 1” (definisci memoria per una parola lunga). Dal momento che il SEKA non conosce questa istruzione, ho dovuto ripiegare su ”dc.l".

Ciò manca completamente nella soluzione Metacomco. Al suo posto vedrete molto spesso all'inizio del listato XDEF. Ciò significa che lo stesso viene definito esternamente (più precisamente nel file “amiga.lib”). Lavorando con il Metacomco, dopo l'assemblaggio è necessario anche un linkaggio. In esso vengono trattati anche i riferimenti esterni. Il SEKA e il DEVPAC possono venire terminati anche senza linkaggio, cioè possono produrre già dei programmi in grado di girare dopo l'assemblaggio.

4.5 Assemblaggio e linkaggio

Dopo che abbiamo battuto questo programma con un Editor, comincia il lavoro vero e proprio. Memorizziamo il testo per es. sotto il nome TEST.S e torniamo al Desktop.
A questo punto dovremo eseguire diversi passaggi che dipendono dal Package di Assembler. Leggiamo quindi il manuale.

Nel caso del SEKA e dell'HiSoft, il tutto è molto semplice. Diamo il comando A oppure Amiga-A per HiSoft, in quest'ultimo caso dovremo anche fare attenzione che sia stato scelto "codice non linkabile", affinché derivi un programma immediatamente eseguibile. Per ottenere ciò sarà possibile inserire come prima riga nel listato "opt l-". Attenzione: deve essere veramente la prima riga, prima di essa non è permessa neanche una riga vuota. Forse è ancora più semplice far girare il programma Install e impostare tramite esso in maniera fissa "opt l-".

Anche nel caso della Metacomco è necessario dapprima assemblare. Qui perché è necessario anche un linkaggio. Dal momento che ambe due hanno bisogno di una lunga sequenza di battitura, sarà opportuno scrivere il seguente file Batch e memorizzarlo sotto il nome make in una Subdirectory.

```
.key f i l e / a
c/assem <file>.s -o <file>.o -c s - i include
c/alink <file>.o to <file> library lib/amiga.lib
```

Battere quindi

```
execute make test (senza .s!!!)
```

Ciò significa: assembla "test.s" e scrivi il risultato nel file codice "test.o". Tralasciando il "-o test.o", avrà luogo solo l'assemblaggio, ma non verrà prodotto nessun file oggetto. Ciò accade molto velocemente ed è consigliabile quando si vuole controllare un programma per verificare se contiene degli errori.

Nel caso in cui non si sia fatto nessun errore (l'assemblatore non ha segnalato nulla) inizierà il linkaggio. Consiglio privato: procuratevi "BLINK" che farebbe parte del DEVPAC, ma che ormai è di dominio pubblico. Questo Linker è molto più veloce di ALINK, e soprattutto in esso manca il cosiddetto "minuto di ripensamento dell'Amiga" (cioè quando il Linker pare non faccia nulla per un lungo periodo di tempo)

4.6 Immissione di stringhe

I testi che noi facciamo uscire possono essere esclusivamente di tipo informativo, ma generalmente ci aspettiamo anche delle immissioni da tastiera. Con la Fig. 4.2 andiamo quindi avanti di un altro passo.

```
        opt     l-                        ;non linkare!

* A2   Il Mio secondo Programma


SysBase            equ 4                  ;Base di Exec
_LVOOpenLibrary    equ -552               ;Apertura della Library
_LVOCloseLibrary   equ -414               ;Chiusura della Library
_LVOOutput         equ -60                ;DOS: Prelevamento dell'Handle di
Output
_LVOWrite          equ -48                ; Output
_LVORead           equ -42
_LVOInput          equ -54

* Apertura DOS/Lib:

_main   move.l  #dosname,a1               ;Nome della DOS-Lib
        moveq   #0,d0                     ;Versione indifferente
        move.1  _SysBase,a6               ;Base di Exec
        jsr     _LVOOpenLibrary(a6)       ;Apertura della DOS-Lib
        tst.l   d0                        ;Errore?
        beq     fini                      ;se errore, fine
        move.1  d0,a6                     ;Annotare il puntatore

* Determinazione Handle di Output:

        jsr    _LVOOutput(a6)             ;Prelevamento Handle di Output
        move.1 d0,d4                      ;e sua annotazione in d4
        move.1 d4,d1                      ;ora emissione del testo
        move.1 #string,d2                 ;come sopra
        moveq  #12,d3
        jsr    _LVOWrite(a6)

* Leggi dalla Tastiera:

        jsr    _LVOInput(a6)              ;Prelevamento Handle di Input
        move.1 d0,d1                      ;e sua copiatura in d1
        move.1 #buffer,d2                 ;Indirizzo del Buffer
        moveq  #80,d3                     ;80 caratteri permessi
        jsr    _LVORead(a6)               ;e lettura
        move.1 d0,len
```

```
* Output del contenuto del Buffer:

        move.l d4,d1                        ;Handle di Output
        move.l #buffer,d2                   ;Indirizzo del testo
        move.l len,d3                       ;Lunghezza del testo
        jsr    _LVOWrite(a6)                ;Funzione "Scrittura"



* Al termine chiudere sempre le Lib!

        move.l a6,a1                        ;Base di DOS-Lib
        move.l _SysBase,a6                  ;Base di Exec
        jsr    _LVOCloseLibrary(a6)         ;Funzione "Chiusura"
fini    rts                                 ;Ritorno al CLI

* Campo dati:

dosname dc.b 'dos.library',0
        cnop  0,2
string  dc.b ‘Enter Text: '
        cnop  0,2
buffer  ds.b  80
len     ds.l  1
```

Fig. 4.2: Input di stringhe

Fino alla riga "* leggi dalla tastiera:" non è cambiato nulla rispetto al primo listato.
Il testo in output è questa volta "Enter Text" ed è a questo punto che l'utente potrà battere sulla tastiera. Per poter sapere da dove vengono gli input, dobbiamo dapprima conoscere gli Handle della tastiera. Analogamente agli Handle di output, possiamo determinare tali Handle con la funzione _LVOInput.

Quando disponiamo degli Handle di input, possiamo chiamare con essi _LVORead.
Questa funzione si comporta praticamente come _LVOWrite, con la sola differenza che in questo caso non daremo più l'indirizzo di un testo, bensì l'indirizzo di un Buffer, nel quale deve venire depositato l'input. Con

```
        buffer    ds.b     80
```

si indica all'Assembler di lasciare uno spazio di 80 Byte. Con "ds.w 40" oppure "ds.l 20" avremmo ottenuto la stessa cosa. Viene riservata in ambedue i casi una parola lunga, nella quale potremo depositare la lunghezza dell'input. La funzione _LVORead è molto flessibile. L'indicazione della lunghezza, con la quale essa viene chiamata, è sempre il massimo. Se si dà meno (fine con tasto di Return) dopo il JSR, la lunghezza reale si

troverà nel Registro d0. Quindi copiamo d0 nella variabile len. Ciò non dovrebbe mai accadere, ma mostriamolo per una volta. Quindi len viene di nuovo copiato nel Registro d3, al fine di chiamare la funzione già nota _LVOWrite, che farà uscire a questo punto il contenuto del Buffer.

4.7 Loop

Nella presente sezione parleremo dei Loop. Dapprima tuttavia vorrei fare qualcosa per la razionalizzazione del nostro lavoro. Abbiamo già battuto due volte la seguente sequenza:

```
SysBase           equ 4                    ;Base di Exec
_LVOOpenLibrary   equ -552                 ;Apertura della Library
_LVOCloseLibrary  equ -414                 ;Chiusura della Library
_LVOOutput        equ -60                  ;DOS: Prelevamento dell'Handle di
Output
_LVOWrite         equ -48                  ; Output
_LVORead          equ -42
_LVOInput         equ -54

* Apertura DOS/Lib:

_main    move.l   #dosname,a1              ;Nome della DOS-Lib
         moveq    #0,d0                    ;Versione indifferente
         move.1   _SysBase,a6              ;Base di Exec
         jsr      _LVOOpenLibrary(a6)      ;Apertura della DOS-Lib
         tst.l    d0                       ;Errore?
         beq      fini                     ;se errore, fine
         move.1   d0,a6                    ;Annotare il puntatore
```

Fig.4.3: Un file Include che verrà usato spesso

Memorizziamo il testo della Fig. 4.3 tramite la funzione di blocco del nostro Editor sotto il nome “OpenDos.i”. Nei programmi che seguono basterà quindi dare una istruzione di Include, e l'Assembler andrà a prendere automaticamente questo testo. Con il Metacomco, il nome del file deve essere contenuto fra virgolette oppure fra apici. Il SEKA non è in grado di effettuare l'Include, e quindi sarà necessario leggere il testo tramite il comando R.

Il seguente compito darà come risultato una stampa di tutte le lettere dalla A alla Z. Cerchiamo di risolverlo in maniera molto primitiva e limitiamoci, per cominciare, alle lettere dalla A alla D.

Anche in questo caso, però, i caratteri devono trovarsi in un Buffer; dobbiamo solo scoprire come metterglieli.

In Fig. 4.4 ho impostato un Registro all'inizio del Buffer, e cioè con

```
        lea.l    buffer,a0
```

ciò significa "carica a0 con l'indirizzo effettivo del Buffer". Il ".l" è superfluo in sé e per sé (gli indirizzi sono sempre lunghi) ma molti Assembler lo esigono.
Il comando ha lo stesso effetto che avrebbe

```
        move.l   #buffer,a0
```

Con l'istruzione

```
        move.b   #'A',(a0)+
```

la costante A viene scritta sull'indirizzo al quale a0 sta puntando (in questo caso inizio del Buffer) ed infine a0 viene aumentato di 1. Un errore tipico è quello di tralasciare il ".b". Siccome gli Assembler assumono sempre per difetto automaticamente ".w" in questo caso a0 verrà aumentato di 2.

Come abbiamo già visto, a0 punterà già all'indirizzo successivo, per cui possiamo continuare con il nostro gioco.

```
        opt      l-                     ;non linkare!

* A3   Il Mio terzo Programma

        include  OpenDos.i

* Determinazione dell'Handle di Output:

        jsr      _LVOOutput(a6)         ;Prelevamento dell'Handle di
Output
        move.l   d0,d4                  ;e sua annotazione in d4


* Riempimento del buffer:

        lea.l    buffer,a0
        move.b   #'A',(a0)+
        move.b   #'B',(a0)+
        move.b   #'C',(a0)+
        move.b   #'D',(a0)+
        move.b   #10,(a0)
```

```
* Ora Output del contenuto del buffer:

        move.l   d4,d1                      ;Handle di Output
        move.l   #buffer,d2                 ;Indirizzo del Testo
        move.l   #5,d3                      ;Lunghezza del Testo
        jsr      _LVOWrite(a6)              ;Funzione "Scrittura"

* Al termine chiudere sempre le Lib!
        move.l   a6,a1                      ;Base di DOS-Lib
        move.l   _SysBase,a6                ;Base di Exec
        jsr      _LVOCloseLibrary(a6)       ;Funzione "Chiusura"

fini    rts                                 ;Ritorno al CLI

* Campo dati:

dosname dc.b     'dos.library',0
          cnop   0,2

buffer             ds.b   80
```

Fig. 4.4: Riempimento Buffer con caratteri – Soluzione 1

4.7.1 Il Loop DBcc

In Fig. 4.4 abbiamo ripetuto sempre le stesse cose, questo è un motivo in più per rivolgerci ad una tecnica più efficace di ripetizione, cioè il Loop. Vogliamo far apparire le lettere dalla A alla Z nello stesso modo in cui le si scriverebbe come programma in BASIC:

```
10 FOR I=ASC("A") TO ASC("Z")
20 PRINT CHR$ (I)
30 NEXT
```

La Fig. 4.5 ci offre la soluzione:

```
        opt     l-                          ;non linkare!

* A4   Il Mio quarto Programma

        include  OpenDos.i

* Determinazione dell'Handle di Output:

        jsr     _LVOOutput(a6)              ;Prelevamento Handle di Output
        move.l  d0,d4                       ;e sua annotazione in d4


* Riempimento del buffer

        lea.l   buffer,a0
        move    #25,d0
        move.b  #'A',d1

loop    move.b  d1,(a0)+
        addq    #1,d1
        dbra    d0,loop
        move.b  #10,(a0)


* e ora emissione del contenuto del buffer:

        move.l  d4,d1                       ;Handle di Output
        move.l  #buffer,d2                  ;Indirizzo del testo
        move.l  #27,d3                      ;Lunghezza del testo
        jsr     _LVOWrite(a6)               ;Funzione "Scrittura"

* Al termine chiudere sempre le Lib!
        move.l  a6,a1                       ;Base di DOS-Lib
        move.l  _SysBase,a6                 ;Base di Exec
        jsr     _LVOCloseLibrary(a6)        ;Funzione "Chiusura"

fini    rts                                 ;Ritorno al CLI

* Campo dati:

dosname  dc.b   'dos.library',0
        cnop    0,2

buffer  ds.b    80
```

Fig. 4.5: Stampa da A a Z con Decc

Contrariamente alla maggior parte dei concorrenti, il 68000 ha un comando di Loop già incorporato, cioè:

```
        DBcc Dn,Etichetta
```

ciò significa “Decrement and Branch on Condition Code” (Decrementa e Salta in caso di Codice di Condizione).

Quindi: con il comando DBCC viene indicato sempre un Registro dati può che essere da D0 a D7, chiamiamolo Dn. Prima dell'ingresso nel Loop, a Dn viene attribuito un valore. Nel Loop, ed esattamente quando il comando DBcc verrà effettuato, Dn verrà decrementato di 1. Finché Dn non diventa -1, ha luogo un salto alla “etichetta”; diversamente verrà eseguito il comando immediatamente successivo. Per quanto concerne “cc”: prima del comando DBcc sarà possibile verificare una condizione per es. con una istruzione CMP (Compare = Paragona) e quindi dire:

```
CMP (A0)+,D0
Dbeq  D1,Etichetta
```

In questo caso il salto all’“etichetta” avrà luogo solo quando la condizione “A(0) eq (equal = uguale) D0” è adempiuta, diversamente il Loop viene terminato. Ciò può venire visto anche da un'altra ottica. Il Loop continuerà a girare per tutto il tempo che la condizione cc non è stata adempiuta, ma comunque al massimo per tutto il tempo in cui il contatore non diventa -1. L'abbreviazione per “cc” è la stessa come per il comando bcc. Per es. esiste BEQ (Branch if Equal = Salta se Uguale) e DBEQ (Decrement and Branch if Equal = Decrementa e Salta se Uguale). Ulteriori dettagli relativamente a tutti i “cc” sono contenuti nel prossimo capitolo, per il momento continuiamo con la pratica. Infatti la condizione di solito non è interessante, si vuole solo contare. In questo caso si dirà semplicemente

DBRA,

che significa Decrement and Branch Always (Salta sempre), naturalmente solo per tutto il tempo in cui il contatore non viene esaurito. Spesso si vede anche “DBF”, dove F significa “False” (falso); ma si tratta solo di un altro modo di scrivere. I buoni Assembler accettano anche “RA” e “F”. Ora possiamo dedicarci al listato di Fig. 4.5. Volevamo stampare le 26 lettere dell’alfabeto dalla A alla Z. Dal momento che il contatore d0 gira sempre fino -1, inizializzando a 25, cioè prima riga. Il codice per le lettere è contenuto nel Registro d1, che viene quindi caricato prima di tutto con “A”.

Ora parte il Loop. Come già detto, mettiamo un segno nel Buffer tramite “(a0) +”.
Ecco la novità: con “addq #1,d1” d1 viene incrementato, per cui A diventa B (B diventa C ecc.). Il lavoro presenta ora la riga successiva:

```
dbra    d0,loop
```

che significa: decrementa d0. Se esso non è ancora -1. salta a ”Loop”, diversamente comando successivo. Nel nostro caso, anche in presenza di -1, si continuerebbe con l'output.

4.8 Le righe di comando

Analizziamo un altro programma che produce un testo. Questa volta però si tratta di un testo che non è definito in nessun punto del programma. E' noto che un programma viene chiamato sotto CLI, battendo il suo nome. E' tuttavia possibile, dopo il nome e dopo un carattere di spazio bianco, inserire del testo a piacere. Questo testo viene chiamato riga di comando. Molti comandi CLI lavorano con esso. Proviamo a battere

```
cd df0:
```

in questo modo si chiama un programma chiamato cd e si immette la riga di comando "df0".

In Fig. 4.6 viene mostrato come viene letto il comando.

```
        opt      l-                       ;non linkare!

* A5   Il Mio quinto Programma

* Prima di tutto salvare l'indirizzo e la lunghezza della riga di testo
        movem.l  a0/d0,-(sp)

        include  OpenDos.i

        jsr      _LVOOutput(a6)           ;Prelevamento dell'Handle di
                         ;Output
        move.l   d0,d1                    ;e la sua destinazione
        movem.l  (sp)+,a0/d0              ;Parametri indietro
        move.l   a0,d2                    ;Indirizzo riga di comando
        move.l   d0,d3                    ;Lunghezza
        jsr      _LVOWrite(a6)            ;Funzione "Scrittura"

* Al termine chiudere sempre le Lib!
        move.l   a6,a1                    ;Base di DOS-Lib
        move.l   _SysBase,a6              ;Base di Exec
        jsr      _LVOCloseLibrary(a6)     ;Funzione "Chiusura"

fini    rts                               ;Ritorno al CLI

* Campo dati:

dosname   dc.b   'dos.library',0
```

Fig. 4.6: Lettura della riga di comando

Il DOS memorizza la riga di comando nella RAM e imposta il Registro a0 sull'indirizzo di partenza, in d0 viene annotata la lunghezza. Dal momento che i Registri d0, d1, a0 e a1 sono, in linea di principio, temporanei (ogni routine li può modificare) un programma che ha bisogno di una riga di comando dovrà dapprima salvare i Registri a0 e d0.

Normalmente ambedue questi Registri vengono memorizzati in variabili, ma vediamo anche un'altra possibilità, e cioè lo Stack.

Il 68000 può mettere o prelevare con un solo comando tutti o alcuni Registri dallo Stack. Con

```
        movem.l  a0/d0,-(sp)
```

vengono messi nello Stack a0 e d0. Sono permesse combinazioni a piacere, quali “a0/a3/a5/d1/d6”, oppure elenchi come “d0-d7/a0-a4” (da d0 a d7, da a0 a a4). L’importante è che in seguito i dati dovranno venire presi dallo Stack con un comando analogo. Nel nostro caso ciò accade con

```
        movem.l (sp)+,a0/d0                ;Parametri indietro
```

Dopo ciò dovremo copiare i Registri nel Registro parametri della funzione Write e potremo emettere la riga di comando. Vedremo in seguito un programma che analizza la riga di comando e che ne trae azioni.

4.9 Sottoprogrammi

Il programma di cui alla Fig. 4.7 dovrà chiedere “Come ti chiami?”. L'utente introdurrà quindi un testo (speriamo il suo nome) e il programma risponderà “Buongiorno caro nome”. Conosciamo già tutto ciò di cui abbiamo bisogno (input e output di stringhe) ma la cosa sta diventando un po’ più pesante. Infatti dobbiamo far uscire un testo tre volte. Naturalmente è possibile scrivere ogni volta l'intera sequenza, ma si è molto più veloci con dei sottoprogrammi. Il problema nel caso di sottoprogrammi è il trasferimento dei parametri. Se io passo tutti e tre i parametri alla funzione Write (3 comandi) e quindi chiamo il mio sottoprogramma “print” non ho guadagnato niente. Potrei passare immediatamente i tre parametri a Write e chiamare tale routine DOS. Di conseguenza possiamo dichiarare:

1. L’Handle dell'output è noto (Si trova in d4)
2. Viene passato solo l'indirizzo del testo. Nel testo la lunghezza è “nascosta”:

Come risolvere tutto ciò e mostrato in Fig. 4.7:

```
        opt      l-                         ;non linkare!

* A6   Mio sesto Programma

        include  OpenDos.i

        jsr      _LVOOutput(a6)             ;Prelevamento dell'Handle di
                         ;Output
        move.l  d0,d4

        lea.l   msg1,a0                     ;richiesta del nome
        bsr     print

        jsr      _LVOInput(a6)              ;Prelevamento dell'Handle di
                         ;Input
        move.l  d0,d1                       ;e sua elaborazione
        lea.l   buffer,a2                   ;Puntatore al buffer
        move.l  a2,d2                       ;trasferimento a Read
        addq.l  #1,d2                       ;salto del byte di lunghezza
        move.l  #79,d3                      ;Il Nome puo' essere lungo
        jsr     _LVORead(a6)                ;Lunghezza reale in d0
        addq.l  #1,d0                       ;ampliare
        move.b  d0,(a2)                     ;e memorizzare
        move.b  #'!',-1(a2,d0.l)            ;! nel buffer
        move.b  #10,0(a2,d0.l)              ;quindi nuova riga

        lea.l   msg2,a0                     ;di' buon giorno
        bsr     print

        move.l  a2,a0                       ;stampa il nome
        bsr     print


* Al termine chiudere sempre le Lib!
        move.l  a6,a1                       ;Base di DOS-Lib
        move.l  _SysBase,a6                 ;Base di Exec
        jsr     _LVOCloseLibrary(a6)        ;Funzione "Chiusura"

fini    rts                                 ;Ritorno al  CLI

print   clr.l    d3
        move.b   (a0)+,d3                   ;Lunghezza
        move.l   d4,d1
        move.l   a0,d2
        jsr      _LVOWrite(a6)              ;Funzione "Scrittura"
        rts

* Campo dati:

dosname  dc.b    'dos.library',0
```

```
        cnop    0,2

msg1    dc.b    16,'Come ti chiami?  '
        cnop    0,2
msg2    dc.b    19,10,'Buongiorno, caro '
        cnop    0,2
buffer  ds.b    80
```

Fig. 4.7: Sottoprogrammi in pratica

Il trucco è nascosto nelle ultime righe. I primi Byte delle stringhe msg1 e msg2 contengono la lunghezza del testo successivo. Dal momento che le stringhe in Pascal vengono memorizzate allo stesso modo, si parla anche di stringhe Pascal.

La chiamata del sotto programma ha luogo secondo il seguente schema:

```
        lea.l   msg1,a0          ;richiesta del nome
        bsr     print
```

Ciò significa che viene passato solo l'indirizzo della stringa corrente. BSR significa "Branch to Subroutine" (Diramazione al Sottoprogramma). La differenza rispetto a JSR è costituita dal fatto che BSR è limitato da una ampiezza di salto di + /-32 Kbyte, mentre JSR vale per l'intero campo di indirizzamento del 68000 (16 Mbyte). Non dovremo quindi usare BSR, che risparmia solo un po' di codice e di tempo. Analizziamo ora il sottoprogramma vero e proprio:

```
print           clr.l   d3
                move.b  (a0)+,d3        ;Lunghezza
                move.l  d4,d1
                move.l  a0,d2
                jsr     _LVOWrite(a6)   ;Funzione "Scrittura"
                rts
```

Nella seconda riga troviamo il nocciolo della questione. Il Byte della lunghezza viene copiato nel Registro d3 (dove aspetta Write). Il piccolo " + " imposta contemporaneamente a0 all'inizio del testo. Due righe più tardi possiamo copiare senza problemi a0 in d2. Funziona, ma l'accesso diretto all'inizio del testo non funzionerà (indirizzo non pari, Write dovrà intercettare questo caso). Ancora a proposito della prima riga: la lunghezza deve venire passata come parola lunga, ma abbiamo solo 1 Byte.

Il nostro problema è che in un Registro ci sono 4 Byte

```
        b3 b2 b1 b0
```

che vengono quindi "moved"

```
move b0
move oppure move.w b1, b0
move.l b3.b2,b1,b0
```

Se passiamo un solo Byte, il resto dei 3 resta invariato. Quindi, nella parola lunga d3 possono trovarsi dei “numeri civici”. Di conseguenza cancelleremo dapprima con clr (Clear = cancella, riempi di zeri) il Registro. D'altra parte, se scriviamo

```
moveq    #1,d3
```

è esatto, in quanto moveq (move quick) amplia automaticamente la costante a “lungo”. D'altra parte la costante è limitata a 8 Bit (-128 fino a +127).

Se ci si è fidati troppo di un determinato procedimento per la trasmissione dei parametri ai sottoprogrammi, ciò potrà avere grosse conseguenze in determinati casi. Per mostrare ciò, proviamo a fare in modo che i testi che vengono letti con Read, vengano fatti uscire con la routine di print.

Come è noto, Read legge in un Buffer partire dal suo inizio. Print invece si aspetta come primo carattere in tale Buffer la lunghezza del testo stesso. Ciò ha per conseguenza:

```
lea.l   buffer,a2         ;Puntatore al buffer
move.l  a2,d2             ;trasferimento a Read
addq.l  #1,d2             ;salto del byte di lunghezza
move.l  #79,d3            ;Il Nome puo' essere lungo
jsr     _LVORead(a6)      ;Lunghezza reale in d0
addq.l  #1,d0             ;ampliare
move.b  d0,(a2)           ;e memorizzare
```

Le prime due righe sono sempre le solite. Impostiamo a2 come puntatore all’inizio del Buffer e copiamo quindi a2 su d2, dove Read normalmente si aspetta l’indirizzo del Buffer. A questo punto però, d2 viene aumentato di 1. Quindi d2 punterà al secondo Byte del Buffer. Read comincerà a riempire il Buffer a partire da questo indirizzo, ed il nostro Byte di lunghezza resta libero. Dopo il JSR, nel Buffer viene copiata anche la lunghezza reale, cosa che viene effettuata da

```
move.b d0,(a2)
```

Osserviamo ora il listato, e vedremo qualcosa in più. Il motivo è che dopo il nome deve venire stampato anche un simbolo di punto esclamativo. Per fare ciò, avremo che a) la lunghezza effettiva verrà aumentata e b) il simbolo di punto esclamativo verrà scritto nel Buffer. Dopo ciò, dovrà esserci solo il 10 (nuova riga). Quindi

```
move.b #’!’,-1(a2,d0.l)       ;! nel buffer
move.b #10,0(a2,d0.l)         ;quindi nuova riga
```

Nel caso in cui il lettore non abbia ancora capito cosa è possibile fare con “ARI con Indice e Offset” (ved. Cap. 3), vediamo qui una applicazione pratica.

Forse la prima cosa che stupisce è il “-1”. E’ opportuno sapere che la funzione di Read memorizza nel Buffer come ultimo carattere anche il tasto di Return (codice ASCII 10) e che di conseguenza lo conta per determinare la lunghezza.

```
(a2,d0.l)
```

significa: costituisci l'indirizzo dalla somma di a2 + d0. Se il Buffer comincia per es. all’indirizzo 1000, e abbiamo battuto i caratteri ABC, nel Buffer avremo:

Indirizzo = 1000	1001	1002	1003
Caratteri = A	B	C	Return

La lunghezza è 4. Di conseguenza 1000 + 4(a2 + d0) = 1004.

Se ora vogliamo porre il simbolo di punto esclamativo all'indirizzo 1003, aggiungeremo ancora l'Offset di -1, cioè sottrarremo 1. Il comando successivo

```
move.b #10,0(a2,d0.l)
```

aggiunge un Offset di zero. Di conseguenza, all’indirizzo 1004 verrà scritto “#10”. Non possiamo tralasciare lo zero. La sintassi del comando vuole avere una costante in quel punto. D’altra parte si incontra spesso questa forma di Offset di zero, in quanto di solito è sufficiente costituire l'indirizzo dai soli due registri.

4.10 Testi segmento di programma, dati e BSS

Spesso troveremo nei listati le seguenti istruzioni:

```
text
data
bss
```

Prima di esse troveremo talvolta la parola SECTION, mentre per “text” incontreremo anche “Code”.

“data” è una istruzione all'assemblatore, di collocare i dati che seguono nel segmento dati del programma. E' importante a questo punto sapere che un programma può essere costituito da segmenti. Il primo segmento si chiama Testo oppure Code, in esso si trova il programma vero e proprio, sarà possibile far cominciare il programma con la parola “text”, mentre per molti assemblatori è addirittura obbligatorio.

Nel segmento dati, si trovano tutti di dati inizializzati cioè quelli che hanno un valore come per es. i nostri testi. In “bss” (block storage segment) vengono collocati i dati che emergono solo durante il periodo di funzionamento del programma. Praticamente si tratta di campi di memoria riservati (originati con l'istruzione DS)

Come detto, è possibile, ma non obbligatorio, formare queste sezioni. Ciò sarà vantaggioso solo per programmi molto lunghi, che offriranno così al caricatore la possibilità di trovare ancora più facilmente dei campi di memoria liberi per i singoli segmenti.
Il DEVPAC, tuttavia, supporta solo una sezione, il SEKA nessuna.

CAPITOLO 5

Diramazioni e menu

“IF THEN”
Bit-Shift
Un po’ di pratica
utilizzo dei tasti di funzione e principio di “ON X GOSUB”

5.1 IF THEN in dettaglio

Avevamo già citato “IF condizione THEN GOTO”, ed ora approfondiamolo un po'.
In linea di principio, essa funziona come nei linguaggi evoluti: Si chiede una condizione e si effettua una diramazione a seconda del risultato. La leggera differenza rispetto ai linguaggi evoluti è che la condizione è lo stato di alcuni Bit nel CCR (Condition Code Register) che fa parte anch’esso del Registro di stato. Tale Registro è mostrato nella figura che segue (Fig. 5.1).

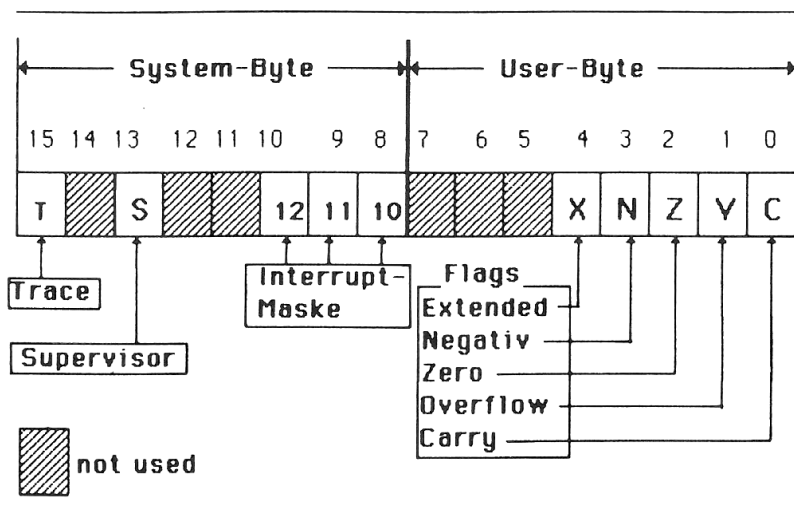



Fig. 5.1: Il registro dello stato del 68000

5.1.1 Il Registro di stato

Vediamo immediatamente che la parola è suddivisa in un Byte di sistema e in un Byte utente. Sappiamo già che il 68000 è in grado di gestire due tipi di funzionamento, cioè in modo supervisore e in modo utente.

Il modo supervisore dovrebbe rimanere riservato al sistema operativo, noi "utenti" lavoreremo, per motivi di sicurezza, solo in modo utente. Quindi dei Bit di cui sopra (chiamati anche Flag) ci interessano solo X, N, Z, V e C.

5.1.2 I Flag

Ci sono molti comandi che influenzano tali Flag, tuttavia si tratta nella maggior parte dei casi di operazioni matematiche con due operandi, dove l'operando sorgente viene sottratto dall'operando di destinazione. Se il risultato e negativo, il 68000 imposterà il Flag N (e il Bit diventa 1).

Se invece si verifica un superamento della capacità, verrà impostato il Bit di Overflow, in caso di riporto (addizione) oppure di "prestito" durante la sottrazione, il Flag di riporto verrà impostato a 1. Se al contrario questi stati non si presentano durante l'operazione, i Flag corrispondenti verranno impostati a zero.

Una ulteriore informazione: T è il Bit di Traccia, S il Bit di Supervisore e i0, i1 e i2 sono la maschera di Interrupt.

5.1.3 Interrogazione dei Flag

	Abbreviazione	Significato	Spiegazione
	CC	Carry Clear	Carry = 0
	CS	Carry Set	Carry = 1
	EQ	Equal	Z = 1
	GE	Greater or Equal	> =
***	GT	Greater Than	>
	HI	Higher	>
***	LE	Less or Equal	<
	LS	Less or Same	< =
***	LT	Less Than	<
	MI	Minus	-
	NE	Not Equal	< >
	PL	Plus	+
***	VC	oVerflow Clear	V = 0
***	VS	oVerflow Set	V = 1

Tab. 5.1: Abbreviazione dei "Condition codes"

E' possibile interrogare i Flag stessi, ma in linea di massima non lo si fa. Al posto di ciò si scrive un comando che influenza i Flag (li controlla) e che effettua se necessario "GOTO Indirizzo, se questo Flag ha tale stato". Facciamo attenzione al fatto che in questo caso GOTO significa diramazione, e per diramazione di solito si scrive B maiuscolo. Ciò che è significativo a questo punto è che (contrariamente a quanto fanno gli "8 Bit"), esistono anche dei comandi di diramazione che tengono conto contemporaneamente di diversi Flag. Un'altra differenza: esistono diversi comandi di diramazione per i numeri privi di segno e per i numeri dotati di segno (complemento di 2). Naturalmente

esistono anche dei comandi che reagiscono solo ad 1 Bit. La tabella 5.1 ne riporta il riassunto.

Gli operatori identificati nella tabella con ***, valgono solo per i numeri in formato di complemento di 2, cioè quelli per i quali il Bit alto serve come segno. I comandi cominciano sempre con B (come Branch = diramazione), seguiti da due lettere, che sono l'abbreviazione della condizione. Se scriviamo per es. BEQ (salta se uguale) dipenderà esclusivamente dal Flag Z se il comando verrà effettuato oppure no.

Il Flag Z può anche essere stato influenzato da un comando che si trovi più o memo lontano dal BEQ. Ora, se sappiamo quale comando influenza il Flag Z è in che modo, possiamo affrontare questo rischio. Naturalmente è molto più sicuro scrivere direttamente prima di BEQ un comando che lo controlli. Se io
per es. voglio saltare quando il Registro D0 è uguale a 0, scriverò:

```
CMP #0,D0
BEQ Etichetta
```

Il comando CMP sottrae dall'operando di destinazione l'operando sorgente, modifica a seconda del risultato il Flag, ma non scrive il risultato nella destinazione. Ciò significa che il comando di confronto ha effetto sul Flag come una sottrazione. Ciò dovrà venire tenuto presente in particolare dagli amici provenienti dagli "8 Bit" quando controlleranno i singoli Flag (con un comando ciascuno). E' inoltre possibile scrivere per es. BGE (salta se maggiore o uguale). Si dovrà far attenzione solo a tre punti:
1. Questi comandi sono efficaci solo direttamente dopo un CMP.
2. Il secondo operando viene confrontato con il primo. Se per es. voglio saltare quando D0 è maggiore di 9 (D0 > 9), scriverò:

```
CMP #9,D0
BGT Etichetta
```

3. Bisognerà inoltre sapere se si è stabilito che gli operandi siano numeri preceduti dal segno oppure privi di segno. Sarà opportuno utilizzare le abbreviazioni anche insieme con DBcc, per es. usando DBMI oppure DBGT. BRA è un caso particolare (salta sempre); ad esso corrispondono anche a DBRA oppure DBF.

5.2 La nostra prima finestra

Con la Fig. 5.2 siamo entrati nella pratica. La funzione Read memorizza un testo in un Buffer carattere per carattere. A questo punto vogliamo sapere quale codice produce un certo tasto. Per fare ciò dovremo stampare il contenuto del Buffer in esadecimale. Il compito principale della Fig. 5.1 è quindi quello di far uscire i Byte sotto forma di due caratteri esadecimali ciascuno. Per una A si dovrebbe stampare per es. 41.

A questo punto devo immediatamente segnalare che è possibile lavorare con i tasti standard sulla console normale, con la quale abbiamo lavorato finora (CON:), ma non con i tasti speciali, quali i tasti di comando del cursore ed i tasti di funzione. Quindi, al fine di poter reagire in un programma con i tasti di funzione, dobbiamo fare qualcosa di particolare.

Dapprima dovremo utilizzare non più gli Handle standard di Input e di Output, ma dovremo fare qualcosa direttamente per l'I/O. L'Amiga ci offre la scelta fra diverse apparecchiature. Fra le altre:

PAR: Interfaccia parallela
SER: Interfaccia seriale
CON: Console
RAW:Console

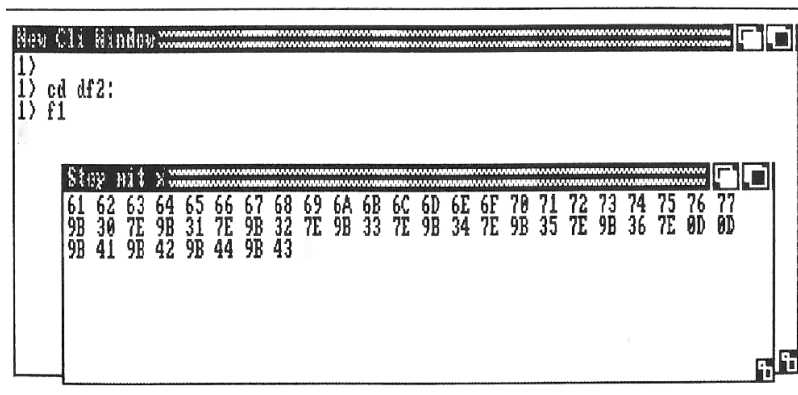



Fig. 5.2 La nostra prima finestra

La differenza fra CON: e RAW: è che solo quest'ultima gestisce tutti i tasti (quindi anche i tasti speciali). Lo svantaggio di RAW: è che tutte le funzioni di modifica sono escluse, cioè dovranno venire impostate direttamente da parte del programma applicativo. RAW: è la tipica apparecchiatura per programmi di Editor.

La Fig. 5.2 presenta il nostro compito. Nella finestra, CLI appare una nuova (nostra) finestra. In questa finestra appaiono sempre dei numeri esadecimali, se battiamo un tasto. I numeri descrivono il codice del tasto. Nella prima riga ho battuto semplicemente le lettere dalla a alla w. Infatti vediamo i codici ASCII corrispondenti da $51 a $77 ($ significa esadecimale).

Nella seconda riga ho azionato i tasti di funzione. Ogni tasto produce tre simboli, cioè:

```
F1        9B 30 7E
F2        9B 31 7E
F3        9B 32 7E
```

etc.

Nella terza riga sono rappresentati i tasti del cursore. Questi generano solamente due simboli, cioè:

```
Su        9B 41
Giù       9B 42
Sinistra  9B 43
Destra    9B 44
```

Vediamo quindi che è comune a tutti i tasti speciali la produzione di una sequenza che comincia con $9B. Ciò accade al fine di poter distinguere immediatamente questi tasti da tutti gli altri. E' quindi necessario valutare immediatamente il simbolo successivo, poiché e solo lì che cominciano le differenze. Rinunciamo per il momento a questa finezza e diciamo semplicemente: visualizzare tutto quello che viene generato. Passiamo quindi al listato di cui a Fig. 5.3

```
                 opt      l-                       ;non linkare!

* F1 Lettura tasto di funzione


        include  OpenDos.i

_LVOOpen         equ  -30
_LVOClose        equ  -36

                 move.l   #name,d1         ;Nome di RAW:
                 move.l   #1005,d2         ;Status = c'è
                 jsr      _LVOOpen(a6)     ;ora apertura
                 move.l   d0,d5            ;annotare l'Handle
                 tst.l    d0               ;Errore?
                 Beq      fini             ;se si, interruzione
```

```
loop            move.l  d5,d1               ;lettura da RAW
                move.l  #buffer,d2          ;in questo Buffer
                move.l  #1,d3               ;1 Carattere
                jsr      _LVORead(a6)       ;chiamata lettura

                cmp.b   #'x',buffer         ;Carattere = 'x' ?
                beq     fertig              ;se si

                move.l  buffer,d2           ;carattere in d2
                lea.l   hbuf,a0             ;Destinazione per la
                         ;trasformazione
                move.b  #' ',2(a0)          ;Spazio bianco dopo esa
                bsr     hex                 ;trasformazione in esadecimale

* Output del numero esadecimale:
                move.l   d5,d1              ;Output anche nella Window
                move.l   #hbuf,d2           ;Indirizzo stringa esadecimale
                move.l   #3,d3              ;Lunghezza
                jsr      _LVOWrite(a6)      ;Funzione "Scrittura"

                bra      loop               ;su di un altro

fertig          move.l   d5,d1              ;Chiudere RAW
                jsr     _LVOClose(a6)

* Al termine chiudere sempre le Lib!
                move.l   a6,a1              ;Base di DOS-Lib
                move.l   _SysBase,a6        ;Base di Exec
                jsr      _LVOCloseLibrary(a6)      ;Funzione "Chiusura"

fini            rts                         ;Ritorno al CLI


* Conversione di d2.l in stringa ASCII ab (a0)
hex
                moveq    #2-1,d1            ;solo per  2 Nibble (di 8)
next            rol.l    #4,d2              ;prelevamento di 1 Nibble
                move.l   d2,d3              ;salvataggio in d3
                and.b    #$0f,d3            ;mascheratura
                add.b    #48,d3             ;trasformazione in ASCII
                cmp.b    #58,d3             ;e' >9 ?
                bcs      out                ;se no
                addq.b   #7,d3              ;diversamente deve essere A-F
out             move.b   d3,(a0)+           ;memorizzazione di 1 Carattere
                dbra     d1,next            ;prossimo nibble
                rts

* Campo dati:

dosname  dc.b   'dos.library',0
                cnop     0,2
name            dc.b     'RAW:40/100/580/80/Stop con x',0
                cnop     0,2
```

```
buffer          ds.b    8
                cnop    0,4

hbuf            ds.b    10
```

Fig. 5.3: Come rilevare i tasti di funzione

Per accedere ad una apparecchiatura accade lo stesso come per un file, cioè l'apparecchiatura deve venire aperta con Open, quindi deve venire richiusa. I DOS-LVO corrispondenti si trovano all'inizio del listato. Per l'apertura è necessario indicare il nome dell'apparecchiatura (del file) ed il modo di accesso. Nei file Include si trovano le uguaglianze

```
MODE_OLDfile EQU 1005
MODE_NEWfile EQU 1006
```

Con Ciò si intende se si deve accedere ad un file già esistente o se se ne vuole aprire uno nuovo. Dal momento che sappiamo (si spera) che RAW: esiste, scegliamo 1005. Alla voce "name" arriva la prima sorpresa:

```
name dc.b 'RAW:40/100/580/80/Stop con x',0
```

Ciò è sufficiente ad aprire l'apparecchiatura RAW:. Contemporaneamente, creiamo una finestra con queste caratteristiche:

40/100: angolo superiore sinistro (x, y in punti dello schermo)
580: larghezza della finestra
80: altezza della finestra
Stop con x: titolo della finestra

La funzione mette a disposizione come al solito un Handle, che potremo utilizzare sia per l'Input che per l'Output, i quali passano, come abbiamo già visto, tramite le funzioni di Read e Write. In caso di Read bisogna fare attenzione. Anche se si fornisce 3 come lunghezza, per i tasti di funzione saranno necessari tre Read.

Dopo la lettura di un carattere, segue

```
cmp.b   #'x',buffer                 ;Carattere = 'x' ?
beq     fertig                      ;se si
```

e con ciò avremo il nostro primo "IF THEN". In chiaro ciò significa: paragona (cmp = compara) la costante x con il primo Byte del Buffer. In caso di coincidenza salta alla

Label “fertig” (branch if equal = salta se uguale). Ciò significherà anche: il nostro programma funziona in Loop, fino a che non viene immessa una x.

Le righe successive preparano la trasformazione in esadecimale. Il sottoprogramma in esadecimale si aspetta una parola lunga in d2. Quindi depositerà il risultato nel Buffer.

5.3 Bit-Shift

Il sottoprogramma mostra chiaramente che spesso in Assembler si deve lavorare a livello di Bit. Con ciò per es. la conversione in esadecimale diventa molto più facile, rispetto al metodo classico (divisione continua per 16). Vedremo anche, non appena se ne presenterà l'occasione, perché la rappresentazione in esadecimale è cosi vantaggiosa. Un esempio: la parola lunga è composta da 4 Byte con il contenuto AA, BB,CC, DD. Vediamo immediatamente che nella parola a valore elevato c’è AABB, mentre nella parola a valore basso c'è CCDD. In decimale sarebbe molto più difficile individuarlo (2864434397).

5.3.1 Convertitore esadecimale

Il problema della routine "esadecimale" è che ora dobbiamo far uscire effettivamente i caratteri ASCII. Se una cifra ha il valore 0,dovremo far uscire il codice ASCII 48 (decimale) al fine di poter vedere lo 0 sul video. Ciò è molto semplice per valori fino a 9 (57), ma per 10 devo stampare in esadecimale “A" e ciò corrisponde al codice ASCII 65, B ha il codice 66 ecc. Dovremo tener presente anche questo buco tra “9” e “A”. Secondo problema: il numero in esadecimale $12345678 ($ significa esadecimale). $1 e un Nibble (mezzo Byte) che occupa nel Registro 4 Bit (0001). Naturalmente dobbiamo indicare dapprima $1, però subito dopo dobbiamo portare $1 nella posizione di $8, poiché nel Buffer il simbolo deve sempre stare davanti (dopo la trasformazione in ASCII-1). Il Byte, però, ha 8 Bit, e, mentre i primi 4 vengono trasferiti, i rimanenti 4 hanno valori che disturbano, per cui li dovremo cancellare. Di conseguenza avremo il seguente flusso:

1. Spostamento del Nibble $1 alla posizione di $8
2. Impostazione a 0 dei 4 Bit rimanenti del Byte
3. Trasformazione del Nibble in ASCII
4. Deposito del carattere nel Buffer
5. Ripetizione dei punti da 1 a 4, con i Nibble $2, $3...$8

> Nota: il sottoprogramma è universale e può convertire anche parole lunghe. Dal momento però che io qui voglio far uscire solo un Byte (il Byte più alto) (2 Nibble) lascerò girare il Loop fine a 2-1.

Il passo 1 viene eseguito con il comando ROL (Rotate Left). Di ROL utilizzeremo la sintassi

```
ROL       #4,d2
```

Ciò significa ruota di 4 Bit verso sinistra il contenuto di d2. E che significa ruotare? Ipotizziamo che in d2 si trovino i seguenti 32Bit

dapprima	1111 0000 0000 0000 0000 0000 0000 0000
dopo ROL #4	0000 0000 0000 0000 0000 0000 0000 1111

Ciò significa che i Bit vengono spostati verso sinistra e che i Bit che si trovavano completamente a sinistra, scomparendo, vengono di nuovo alimentati da destra. Sarebbe diverso in caso di comando ASL (Shift left, sposta verso sinistra). Anche in quel caso si avrebbe uno spostamento verso sinistra, però a destra verrebbero aggiunti solo degli zeri e a sinistra i Bit che c'erano scomparirebbero in ogni caso, abbiamo scelto ROL, e ciò è sufficiente al nostro scopo. Il Nibble (nome di un mezzo Byte) che si trovava in precedenza completamente a sinistra, si trova ora completamente a destra.

Il nostro Loop dovrà girare due volte, quindi dovrò caricare il contatore di Loop D1 con uno (a causa del -1)(2-1). Ora ha luogo il ROL. Dopo ciò il Nibble si troverà esattamente, nel posto giusto, ma non è possibile far uscire questi 4 Bit da soli, abbiamo bisogno di un Byte per un carattere ASCII, quindi di 8 Bit.

5.3.2 Maschere

I 4 Bit che si trovano a sinistra del mio Nibble hanno un valore che deve venire cancellato (più precisamente: i 4 Bit più alti del Byte devono diventare 0000). Ciò accade tramite la maschera $F, con "andi.b #$0F,d3". Esempio:

in d3 abbiamo	1011 1010
Maschera AND	0000 1111
risultato	0000 1010

Questo procedimento è possibile per il fatto che la E logica (AND) si realizza solo quando tutte le grandezze in ingresso sono vere. In Assembler, la E (AND) ha effetto Bit per Bit. Dopo aver isolato il valore numerico puro (da 0 a 15 in decimale), ha inizio la conversione in ASCII. Da 0 a 9 (come numeri) possono venire convertiti direttamente in 0-9 (come caratteri ASCII),cioè nei codici ASCII da 48 a 57. Ciò significa però che dobbiamo per lo meno aggiungere 48. Ora paragoniamo D3 con 58 (cioè un 10 in esadecimale, che bisogna indicare con A). Se il numero è minore di 10 (cioè da 0 a 9) può rimanere com'è e procedere. Diversamente dovremo aggiungere la differenza di cui alla tabella ASCII (65 per A - 58 = 7).

A questo punto è opportuno un consiglio. Spesso si cerca di controllare se un valore si trova fra 0 e 9, quindi, con un secondo test, si controlla se si trova fra 10 e 15. Questo metodo va bene se i caratteri provengono da un Input, nel quale l'utente può battere

anche dei caratteri non validi (non 0...9, A...F). In questo caso, invece, ci procuriamo i numeri da un Registro, dove non è possibile che ci siano caratteri non validi.

I comandi CMP e BCS sono già stati analizzati. Perché BCS? Sui Flag, un paragone ha lo stesso effetto di una sottrazione, in caso di un “prestito” viene impostato anche il Flag di prestito, ciò accade quando D3 è maggiore di 58.

5.4 La diramazione multipla

E' noto che premendo un tasto di funzione, un programma farà qualcosa di particolare, cioè chiamerà una routine. Può trattarsi di parti di programma molto complicate, ma esercitiamoci sul principio come segue

```
IF F1 THEN GOTO ...
IF F2 THEN GOTO
ecc.
```

La routine chiamata potrà anche rispondere semplicemente “qui F1, 2 ecc.”, quindi non ci sarà molto da battere, semplicemente si termina ad F4. La Fig. 5.4 mostra la soluzione.

```
            opt l-          ;non linkare!

* F2 Lettura del tasto di funzione F2 e azione


        include  OpenDos.i

_LVOOpen         equ  -30
_LVOClose        equ  -36
_LVOWaitForChar  equ  -204

        move.l   #name,d1                  ;Nome di RAW:
        move.l   #1005,d2                  ;Status = c'è
        jsr      _LVOOpen(a6)              ;ora apertura
        move.l   d0,d5                     ;annotare l'Handle
        tst.l    d0                        ;Errore?
        Beq      fini                      ;se si, interruzione

        lea.l    buffer,a3                 ;Indirizzo del Buffer

loop    jsr      GetKey                    ;Lettura tasto
        cmp.b    #$9B,(a3)                 ;Tasto di Funzione?
```

```
        Bne     loop                    ;se no
        jsr     GetKey                  ;altrimenti lettura codice
        cmp.b   #$30,(a3)               ;Tasto F1 ?
        beq     F1                      ;se si
        cmp.b   #$31,(a3)               ;       F2 ?
        beq     F2                      ;se si
        cmp.b   #$32,(a3)               ;       F3 ?
        beq     F3                      ;se si
        cmp.b   #$33,(a3)               ;Tasto F4
        beq     F4                      ;se si, fine
        bra     loop

F1      lea.l   f1_text,a0              ;Indirizzo di testo
        bsr     print                   ;stampa
        bra     loop                    ;un nuovo tasto

F2      lea.l   f2_text,a0              ;Indirizzo di testo
        bsr     print                   ;stampa
        bra     loop                    ;un nuovo tasto

F3      lea.l   f3_text,a0              ;Indirizzo di testo
        bsr     print                   ;stampa
        bra     loop                    ;un nuovo tasto


F4      move.l  d5,d1                   ;Chiusura di RAW
        jsr     _LVOClose(a6)

* Al termine chiudere sempre le Lib!
        move.l  a6,a1                   ;Base di DOS-Lib
        move.l  _SysBase,a6             ;Base di Exec
        jsr     _LVOCloseLibrary(a6)    ;Funzione "Chiusura"

fini    rts                             ;Ritorno al CLI


GetKey  move.l  d5,d1                   ;Lettura di RAW
        move.l  a3,d2                   ;in questo Buffer
        move.l  #1,d3                   ;1 Carattere
        jsr     _LVORead(a6)            ;Chiamata di lettura
        rts

print   clr.l   d3
        move.b  (a0)+,d3                ;Lunghezza
        move.l  d5,d1
        move.l  a0,d2
        jsr     _LVOWrite(a6)           ;Funzione "Scrittura"
        rts

* Campo dati:

dosname dc.b    'dos.library',0
        cnop    0,2
```

```
name     dc.b     'RAW:40/100/580/80/Stop con F4',0
         cnop     0,2

f1_text  dc.b     8,'Qui F1',10
         cnop     0,2
f2_text  dc.b     8,'Qui F2',10
         cnop     0,2
f3_text  dc.b     8,'Qui F3',10
         cnop     0,2

buffer   ds.b     8
         cnop     0,4

hbuf     ds.b     10
```

Fig 54: Diramazioni con molti IF THEN

5.5 Soluzione 1: molti IF THEN

Non c'è bisogno di una grossa spiegazione per questo programma, ma ogni programma ha un proprio significato. In casi di emergenza può servire anche come esempio deterrente. Immaginiamoci come funzionerebbe un programma con 100 comandi o più!

Qui la novità è che controlliamo immediatamente $9B, ed in caso di “nessun tasto speciale”, aspettiamo immediatamente il carattere successivo. Siccome però nel caso $9B dobbiamo leggere ancora un tasto, trasferiamo la lettura nel sotto programma “GetKey”.

5.6 Soluzione 2: ON X GOSUB in Assembler

Tutto ciò può venire risolto in una maniera ancora più elegante con un “ON X GOSUB”. Il livello di difficoltà, in questo caso, aumenta un po’, ma non appena lo avremo imparato, saremo già a buon punto. Ogni programma è costituito infatti da un Loop principale, nel quale esso aspetta i comandi. I comandi vengono interpretati e vengono chiamati i sottoprogrammi necessari. In BASIC, avremmo:

```
10 INPUT COMMAND
20 ON COMMAND GOSUB 100,200,300,...
30 GOTO 10
```

in Fig. 5.5, vediamo invece com'è in Assembler.

```
        opt     l-                              ;non linkare

* F3    ON Tasto di funzione GOSUB


    include     OpenDos.i

_LVOOpen        equ  -30
_LVOClose       equ  -36
_LVOWaitForChar equ  -204

                move.l  #name,d1                ;Nome di RAW:
                move.l  #1005,d2                ;Status = c'è
                jsr     _LVOOpen(a6)            ;ora apertura
                move.l  d0,d5                   ;annotare l'Handle
                tst.l   d0                      ;Errore?
                beq     fini                    ;se si, interruzione

                lea.l   buffer,a3               ;Indirizzo del Buffer

loop            jsr     GetKey                  ;Lettura tasto
                cmp.b   #$9B,(a3)               ;Tasto di Funzione?
                bne     loop                    ;se no
                jsr     GetKey                  ;altrimenti lettura
                        ;codice (Code)

                move.b  (a3),d0                 ;Code -> d0
                ext.w   d0                      ;ampliamento a parola
                sub.w   #$30,d0                 ;Code in 0..3
                asl.w   #2,d0                   ;per 4
                lea.l   table,a0                ;Puntatore a Tabella
                move.l  0(a0,d0.w),a0           ;Indirizzo -> a0
                jsr     (a0)                    ;Chiamata delle Routine

                bra     loop                    ;finche' non arriva F4


F1              lea.l   f1_text,a0              ;Indirizzo testo
                bsr     print                   ;stampa
                rts

F2              lea.l   f2_text,a0              ;Indirizzo testo
                bsr     print                   ;stampa
                rts

F3              lea.l   f3_text,a0              ;Indirizzo testo
                bsr     print                   ;stampa
                rts

F4              move.l  (sp)+,d0                ;Kill Return Address
                move.l  d5,d1                   ;Chiusura di RAW
```

```
                jsr     _LVOClose(a6)

*  Al termine chiudere sempre le Lib!
                move.l  a6,a1                   ;Base di DOS-Lib
                move.l  _SysBase,a6             ;Base di Exec
                jsr     _LVOCloseLibrary(a6)    ;Funzione "Chiusura"

fini            rts                             ;Ritorno al CLI

GetKey          move.l  d5,d1                   ;Lettura di RAW
                move.l  a3,d2                   ;in questo Buffer
                move.l  #1,d3                   ;1 Carattere
                jsr     _LVORead(a6)            ;Chiamata di lettura


                rts

print           clr.l   d3
                move.b  (a0)+,d3                ;Lunghezza
                move.l  d5,d1
                move.l  a0,d2
                jsr     _LVOWrite(a6)           ;Funzione "Scrittura"
                rts

* Campo dati:

table           dc.l    F1
                dc.l    F2
                dc.l    F3
                dc.l    F4

dosname         dc.b    'dos.library',0
                cnop    0,2
name            dc.b    'RAW:40/100/580/80/Stop con F4',0
                cnop    0,2

f1_text         dc.b    8,'Qui F1',10
                cnop    0,2
f2_text         dc.b    8,'Qui F2',10
                cnop    0,2
f3_text         dc.b    8,'Qui F3',10
                cnop    0,2

buffer          ds.b    8
                cnop    0,4

hbuf            ds.b    10
```

Fig. 5.5: ON X GOSUB in Assembler

L'inizio del listato è già quasi completamente noto. Qui, il codice del tasto di funzione (il Byte dopo $9B) viene trasformato, sottraendo $30, nei numeri da 0 a 9: per cui si troverà nel Registro D0. Per semplicità, lavoriamo anche qui solo con i tasti di funzione F1, F2, F3 ed F4, cioè 0, 1, 2 e 3 in D0. Facciamo attenzione che mancano tutti i test (parliamo solo del principio), per cui in caso di azionamento di tutti gli altri tasti,il programma verrà interrotto. Vogliamo realizzare un “ON D0 GOSUB” e per fare ciò abbiamo prima di tutto bisogno di 4 sottoprogrammi, che si chiameranno di nuovo F1, F2, F3 ed F4. Questi sotto programmi sono banali. Essi forniscono solo la segnalazione “qui F1” ecc. Osserviamo ora il listato dal basso. Troviamo la label “table” e proprio questo è il segreto del nostro “ON X GOSUB”.

Per le diramazioni multiple, nell’Assembler c'è bisogno di una tabella di salto. Tale tabella è un elenco con gli indirizzi dei sottoprogrammi. La determinazione della tabella è molto semplice. Per ogni indirizzo scriviamo

```
        dc.l     Label
```

dove per Label deve venire introdotta la marcatura (l'indirizzo simbolico) dei singoli sottoprogrammi. La cosa importante è la sequenza, in questo caso abbiamo l'attribuzione

Tasto	Routine
Tasto di funzione l	F1
Tasto di funzione 2	F2
Tasto di funzione 3	F3

La sequenza dei tasti dovrà riflettersi nella sequenza delle immissioni (le si chiama cosi) della tabella. Anche i sottoprogrammi possono trovarsi nel listato in una sequenza a piacere. Esiste tuttavia una attribuzione molto stretta fra i comandi (in questo caso i tasti di funzione) e la tabella di salto. Di conseguenza è possibile calcolare dai comandi gli Indirizzi corrispondenti. Un indirizzo occupa sempre 4 Byte, per cui la nostra tabella potrà trovarsi in memoria come segue:

Label	Indirizzo
Fl	1000
F2	1004
F3	1008

Il nostro comando (in D0) potrà essere:

Valore	0,	1	oppure 2
che per 3 dà	0,	4	oppure 8
e che più 1000 dà	1000,	1004	oppure 1008

E' proprio cosi semplice! Moltiplichiamo ora D0 per 4. Per fare ciò il 68000 ha naturalmente anche un comando speciale (MULU) ma per il momento non lo utilizzeremo. Un programmatore in Assembler esperto, per risolvere le moltiplicazioni per 2, oppure 4, oppure 8, oppure 16 (notare: sempre potenze di 2) utilizzerà immediatamente un comando che effettua questo conto in maniera più veloce. In questo caso tale comando si chiama

```
ASL     Arithmetic Shift Left
```

“ASL #1,d0” per es., sposta di una posizione verso sinistra tutti i Bit in D0. L’effetto è praticamente lo stesso come nel sistema decimale, dove si moltiplica per 10 spostando verso sinistra i numeri (e tenendo ferma la virgola). Qui, però, ci troviamo in un sistema numerico binario, dove è possibile solo una moltiplicazione per 2. Se però spostiamo di due posizioni, otteniamo il risultato di una moltiplicazione per 4. Tale risultato dovrà venire aggiunto all'inizio della tabella. Il suo indirizzo di partenza è ottenibile tramite “lea table,a0”.
Ed ecco ora un comando molto strano, cioè

```
move.l 0(a0,d0.w),a0
```

Utilizziamo il tipo di indirizzamento “ARI con Indice Offset”. Solo che “Indice” non è possibile, per cui impostiamo l’Offset a 0. Dopo ciò si calcola l'indirizzo effettivo come somma di a0 e d0. Ora è sufficiente “move” per portare l'indirizzo nel Registro di destinazione, e prendere di nuovo a0. Ciò è permesso per il 68000: a0 punta ora all'indirizzo del sotto-programma corrispondente, che noi potremo chiamare semplicemente con “jsr (a0)”. Dopo il “jsr” il programma ritorna al “jsr” del comando successivo. Quest’ultimo è “bra loop” per cui si riparte di nuovo con il comando successivo.

L'eccezione a questa regola si trova nel sotto-programma “F4”, Questo sotto-programma in pratica non è nulla, esso infatti viene chiamato con JSR, ma non termina con RTS. Di conseguenza dobbiamo togliere l'indirizzo di Return che si trova ancora nello Stack. Per spiegare ciò, gli americani dicono semplicemente “Kill Return Address” (Uccidi l’Indirizzo di Return), che noi effettueremo con il comando “move.l (Sp)+,d0”. Ciò è permesso, perché in questo caso D0 non è più necessario. Ha comunque preso piede anche la seguente sintassi:

```
move.l (sp)+,(sp)
```

In questo modo l'indirizzo di Return viene prelevato dallo Stack e riscritto immediatamente sullo Stack. A causa del “ + ” anche il puntatore allo Stack è cambiato; quindi abbiamo raggiunto lo scopo. Naturalmente sarebbe corretto anche un “addq.l #4.sp” ma ciascuno ha il proprio metodo particolare per produrre una azione di “Uccisione dell'indirizzo di Return”.

5.7 Soluzione di CASE X OF

Semplice, vero? Nel caso fosse troppo semplice, rendiamo la cosa più complicata.
Il nostro interprete dei comandi ha uno svantaggio. I comandi devono presentarsi nella sequenza dei codici dei tasti di funzione. Anche altre sequenze, come da 1 a 9 oppure da A a M sarebbero possibili, ma deve comunque sempre trattarsi di una sequenza.
Ecco perché molti offrono un menu come segue

1 = Input
2 = Calcolo
3 = Stop

E' difficile da ricordare, sarebbe molto meglio

I = Input
C = Calcolo
S = Stop

5.8 Come lavorare con due tabelle

Il principio è naturalmente molto facile. Ci sono due tabelle. Nella prima tabella si trovano le “Keys” (i tasti o i comandi permessi) e nella seconda si trovano gli indirizzi delle routine corrispondenti. Si dovrà quindi cercare solo il tasto della prima tabella e, a seconda del suo numero di posizione in tale tabella, determinare un puntatore alla posizione giusta della tabella degli indirizzi. A questo punto si potrà prelevare l'indirizzo e continuare.

Questa volta però il programma dovrebbe essere veramente perfetto. Di conseguenza sarà importante intercettare il caso di “non trovato”. Inoltre non vogliamo costringere l'utente ad utilizzare sempre il tasto delle maiuscole, di conseguenza tali lettere maiuscole e minuscole dovrebbero venire gestite indifferentemente. In conclusione il programma dovrebbe essere universale, cioè: dovrebbe essere possibile con il minimo dello sforzo aggiungere o modificare funzioni.

Osserviamo quindi la Fig 5.6 con il listato.

```
        opt l-                              ;non linkare

* F4    CASE X OF


                 include  OpenDos.i

_LVOOpen         equ      -30
_LVOClose        equ      -36

                 move.l   #name,d1          ;Nome di RAW:
                 move.l   #1005,d2          ;Status = c'è
                 jsr      _LVOOpen(a6)      ;ora apertura
                 move.l   d0,d5             ;annotare l'Handle
                 tst.l    d0                ;Errore?
                 beq      fini              ;se si, interruzione


loop             jsr      GetKey            ;Lettura tasto

* Ricerca del tasto nella Tabella dei tasti validi
* ------------------------------------
                 move.b   (a3),d0           ;Code -> d0
                 bclr     #5,d0             ;Forza le maiuscole
                 lea      keys,a0           ;Tab. tasti validi
                 move     #count,d1         ;loro numero
search           cmp.b    (a0)+,d0          ;Tasto qui?
                 dbeq     d1,search         ;se no
                 tst      d1                ;Tasto trovato?
                 bmi      loop              ;se no

* Calcolo indirizzo per il Tasto
* ---------------------
                 neg      d1                ;sub d1,#count
                 add      #count,d1         ;fornisce il n.ro di posiz.e del
tasto
                 lsl      #2,d1             ;che per 4
                 lea      table,a0          ;Puntatore alla Tabella
                 move.l   0(a0,d1),a0       ;Indirizzo -> a0

                 jsr      (a0)              ;chiamata della Routine

                 bra      loop              ;finche' arriva F4


Eingabe          lea      E_text,a0         ;Indirizzo testo
                 bsr      print             ;stampa
                 rts

Rechnen          lea      R_text,a0         ;Indirizzo testo
                 bsr      print             ;stampa
                 rts


Stoppen          move.l   (sp)+,d0          ;Kill Return Address
```

```
                move.l  d5,d1           ;Chiusura di RAW
                jsr     _LVOClose(a6)

* Al termine chiudere sempre le Lib!
                move.l  a6,a1           ;Base di DOS-Lib
                move.l  _SysBase,a6     ;Base di Exec
                jsr     _LVOCloseLibrary(a6)     ;Funzione "Chiusura"

fini            rts                     ;Ritorno al CLI

GetKey          move.l  d5,d1           ;Lettura di RAW
                move.l  a3,d2           ;in questo Buffer
                move.l  #1,d3           ;1 Carattere
                jsr     _LVORead(a6)    ;Chiamata di lettura
                rts

print           move.l  a0,a1           ;copiatura Puntatore al testo
p1              addq.l  #1,a1           ;+1
                cmp.b   #0,(a1)+        ;Byte di zero?
                bne     p1              ;se no
                sub.l   a0,a1           ;= Lunghezza testo

                move.l  a1,d3           ;Lunghezza
                move.l  d5,d1           ;Handle
                move.l  a0,d2           ;Indirizzo testo
                jsr     _LVOWrite(a6)   ;Funzione "Scrittura"
                rts

* Campo dati:

table           dc.l    Input
                dc.l    Calcolo
                dc.l    Stop

keys            dc.b    'I','C','S'
count           equ     *-keys
                cnop    0,4

dosname         dc.b    'dos.library',0
                cnop    0,2
name            dc.b    'RAW:40/100/580/80/Stop con S',0
                cnop    0,2

E_text          dc.b    'Qui Input',10,0
                cnop    0,2
R_text          dc.b    'Qui Calcolo',10,0
                cnop    0,2

buffer          ds.b    8
                cnop    0,4

hbuf            ds.b    10
```

Fig 5.6: Tecnica universale di menu

Questa volta cominciamo dall'inizio. Troviamo un tasto, ma poiché non ci aspettiamo un tasto di funzione, lo ignoriamo.

Maiuscola forzata

Dopo che il carattere è stato caricato nel Registro D0, esso deve venire trasformato in lettera maiuscola, nel caso in cui non sia già una lettera maiuscola. Gli specialisti chia- mano ciò “Force Uppercase”.

Se osserviamo ora una tabella ASCII, noteremo immediatamente che le lettere maiusco- le e minuscole si distinguono l'una dall’altra sempre per 32. Però, anche 2 alla quinta è 32, cioè, in caso di lettere minuscole, sarà impostato il Bit 5. Di conseguenza un buon programmatore in Assembler non dirà “se il codice è maggiore di Z, sottrai 32”, ma dirà semplicemente “cancella il Bit 5”. Per fare ciò,il 68000 ha un suo comando, cioè “BCLR” (Bit Clear)

```
bclr #5,d0
```

cancella il Bit 5 (lo mette a 0) nell’operando (in questo caso d0).

Ora siamo già al punto di poter controllare, se l’utente ha immesso I, C oppure S. Osserviamo per un attimo la fine del listato troveremo una piccola tabella per questi tre caratteri.

5.9 Contatori di posizioni ed uguaglianze

Sotto queste lettere troviamo

```
count equ *-keys
```

Cominciamo ad analizzare “equ”. Equ corrisponde al cosiddetto Equate (in inglese) che per noi significa uguaglianza.

L'istruzione in Assembler per es.

```
Antonio equ 4711
```

significa che a partire da ora si potrà dire Antonio al posto di 4711. A questo punto “JSR Antonio” sarebbe un comando valido. In linea di principio, ciò non è molto diverso da una elaborazione di testo. Infatti, in seguito, l'Assembler metterà al posto di Antonio, un 4711. Dobbiamo però vedere Antonio anche come costante. Nel caso in cui si conosca il

Pascal, si consideri “equ” come “const”, mentre, come programmatori in C, come un “#define”. Di conseguenza nel presente programma non potremo scrivere “count”, facendo riferimento a “count”, per noi è obbligatorio scrivere “ #count”.

Un'altra sorpresa è il “*”. Come primo simbolo in una riga di programma esso è assolutamente inoffensivo e significa solo “segue commento”. Come operando, esso significa contatore di posizione (LC). E' noto che i comandi occupano uno spazio in memoria molto diverso a seconda del numero e del tipo degli operandi. Di conseguenza l’Assembler tiene un contatore, che conta per così dire i Byte utilizzati per ogni comando. Da questo punto di vista, l'LC corrisponde al PC (Contatore di Programma) con il quale la CPU insegue un programma. La differenza è che l'LC viene incrementato anche con istruzioni in Assembler, come per es. “dc.b” che occupa dei Byte (con dei dati).

Nella sintassi Assembler, l'LC non si chiama LC, bensì “*”. Osserviamo un esempio. Nel listato, l'LC che si trova all'inizio della riga con la label “keys” avrebbe il valore di 100. L'istruzione “dc.b I, C, S” lo lascia a 100 (per I), poi lo porta a 101 (per C) ed infine a 102 (per S).

La riga successivo con la label “count” vede l'LC come 103. A questo punto ha luogo una uguaglianza. Dal momento che nelle uguaglianze sono permesse anche delle espressioni, il nostro

```
count equ *-keys
```

avrà l'effetto di

```
count = LC - keys
```

Ciò darà in numeri

```
count = 103-100
```

Con ciò avremmo quindi la nostra costante simbolica 3. Perché non abbiamo scritto immediatamente "count equ 3"? Risposta: lo fanno solo i principianti!

Se in seguito ampliamo la tabella, potremo farlo senza occuparci delle righe con l’ “equ”. In ogni procedimento di assemblaggio verrà introdotta automaticamente la cifra giusta. Inoltre le tabelle possono essere molto lunghe, quindi è facile sbagliarsi.

5.10 Ricerca con DBcc

Ora dobbiamo cercare il tasto (che si trova sempre in D0) nella tabella “keys”. Ciò è contenuto sotto la voce “ricerca di codici tasto in tabella”. Il problema è che anche “non trovato” deve venire riconosciuto quando viene segnalato. La soluzione è un Loop DBcc. Teniamo sempre presente:

```
DBcc dn.loop
```

esso significa; abbandona il Loop, se la condizione cc è adempiuta, oppure se dn è andato a -1, diversamente salta al “loop”.

Prima però è stato impostato un puntatore all'inizio della tabella con “lea keys,a0”. Il trucco è che il contatore di Loop D1 verrà caricato con count (in questo caso 3), mentre un Loop DBcc gira fino a -1, in questo caso quindi, quattro passi avanti. Questo significa che quando il Loop, a causa di “equ” (“Equal” significa trovato) viene lasciato in “dbeq”, D1 non può essere negativo. Se è negativo, avrà effetto “bmi start” (salta se negativo).

Con “cerca indirizzo per tasto” emerge il problema successivo. Nel Loop Dbcc,D1 andava all’indietro. Quindi, in caso di “trovato”, esso assumerà questi valori:

Key	D1
1	3
2	2
3	1

Al fine di poter accedere alla tabella indirizzi come nell'esempio precedente, abbiamo bisogno della sequenza 0, 1, 2. Ciò potrebbe accadere molto semplicemente se io sottraessi D1 da Count (3 - 3 = 0, 3 - 2 = 1, 3 - 1 = 2). Purtroppo il 68000 non permette il comando “sub d1,#count”; e allora come potrà sottrarre qualcosa da una costante?

Ci aiuta l'istruzione

```
neg dl
```

Io nego d1. Se esso era 3, diventerà -3. A ciò aggiungo Count risultato 0. Vecchia regola: se non si vuole sottrarre, sarà necessario aggiungere un valore negativo. Il resto era già noto. L'unica differenza è che io qui invece di “asl” ho preso “lsl” (“logical shift left”). La differenza è che ASL shifta in modo aritmeticamente corretto, e quindi terrebbe conto anche del segno, se ce ne fosse uno. Con ciò,ci troveremmo di nuovo nel punto dove eravamo arrivati con il programma precedente. Con “lea table,a0” puntiamo alla tabella indirizzi, il resto è già noto.

Un'occhiata ai testi mostra immediatamente che ho tralasciato i Byte di lunghezza. Al posto di tali Byte tutti i testi sono stati terminati con un Byte 0. Questa tecnica è molto pratica, dal momento che ora non voglio occuparmi assolutamente della lunghezza dei testi (conteggio dei caratteri). Ma il sottoprogramma dovrà fare qualcosa di più.

Sono state aggiunte le seguenti righe;

```
print   move.l   a0,a1            ;copiatura Puntatore al testo
p1      addq.l   #1,a1            ;+1
        cmp.b    #0,(a1)+ ;Byte di zero?
        bne      p1               ;se no
        sub.l    a0,a1            ;= Lunghezza testo
```

Nel piccolo Loop si cerca semplicemente il Byte di zero. La routine non trova nessun caso nel quale la stringa sia vuota (cioè sia costituita solo da un Byte di 0).

Se non lavoriamo con dei sottoprogrammi, possiamo mettere a disposizione la lunghezza del testo anche tramite il contatore di posizione. Otterremmo quindi:

```
Text_1  dc.b     'Questo è testo'
Len_1   equ      *-Text_1
```

L'output di questo testo sarebbe quindi per es.:

```
move.l  d5,d1           ;Handle
move.l  #Text_1,d       ;Indirizzo testo
moveq   #Len_1,d3       ;Lunghezza
jsr     _LVOWrite(a6)   ;Funzione "Scrittura"
```

Osserviamo ancora: "moveq" amplia automaticamente ad una parola lunga, ma delimita la lunghezza del testo a 127 caratteri. Di conseguenza sarà opportuno utilizzare il "move.l" anche se è un po' più lento.

CAPITOLO 6

Razionalizzazione del lavoro

Strutturazione dei programmi in Assembler
Macro
File “Include”
Moduli

Sapete esattamente, quante righe di programma un programmatore professionista scrive al giorno? 6 (in parola sei) righe al giorno! Si ottiene infatti tale numero come risultato, se si sommano fra loro i giorni impiegati a partire dall'analisi del problema, passando per le prime bozze preliminari, la programmazione vera e propria, il collaudo, il Debugging, fino alla documentazione e se poi si divide il numero di righe di programma per tale numero di giorni.

In tutto ciò, il linguaggio di programmazione impiegato non ha nessuna importanza. Risultano sempre queste sei righe.

Naturalmente con sei righe in Pascal si potrà ottenere un effetto molto maggiore, di quanto non si ottenga con sei righe in Assembler. Inoltre è importantissimo, proprio nella programmazione in Assembler, poter sfruttare tutte le possibilità volte alla razionalizzazione del lavoro.

6.1 Strutturazione di programmi in Assembler

Il metodo più efficace è la strutturazione dei programmi. Con ciò non si intende assolutamente “WHILE...WEND” e simili (che tutta via sono utili, ved. 6.1.1) bensì la struttura di base del programma. In generale, un programma si comporta come segue:

Logo
-Menu
-Attesa degli Input
-Reazione agli Input

Con Logo si intende quell'immagine che appare sullo schermo al lancio del programma. Dopo ciò , le funzioni possibili del programma vengono offerte all'utente in un menu (principale) con attesa di Input. L'Input viene interpretato e a seguito di ciò ,viene chiamata la funzione corrispondente. Abbiamo già esercitato nella pratica questa tecnica, al Capitolo 5. In BASIC, avremmo:

```
100 PRINT "LOGO"
200 PRINT "MENU"
300 INPUT KEY
400 ON KEY GOSUB 1000,2000,3000,….
410 GOTO LOGO
```

In Assembler non abbiamo grosse variazioni di base. Con i comandi tipici di questo linguaggio, si potrebbe scrivere: (“bsr” significa “Branch to Sub Routine”, quindi GOSUB).

```
                bsr logo
        loop    bsr menu
                bsr input               ; Attesa di input
                bsr calcola_indir.
                bsr indirizzo           ; Chiamata funzione
                bra loop
;
;Inizio dei sottoprogrammi
```

E' evidente che l'applicazione essenziale è l'articolazione in sottoprogrammi. Questi sottoprogrammi chiamano anch'essi dei sottoprogrammi, e anche i "sotto sottoprogrammi" chiamano ulteriori sottoprogrammi che possono essere per es. solo funzioni del sistema operativo. Proprio queste ultime presentano tuttavia alcune caratteristiche importanti, che i sottoprogrammi devono avere, cioè:

- utilizzabilità universale
- interfaccia univoca chiaramente definita.

Un buon esempio sono le routine descritte precedentemente. E' infatti possibile trovarsi a dover offrire a determinati punti del menu principale, dei menu secondari. Quindi sarà molto pratico poter utilizzare anche per essi gli stessi sottoprogrammi.

6.1.1 Struttura del linguaggio

Nel Capitolo 4 abbiamo visto un esempio che stampava le lettere dalla A alla Z. Osserviamo ancora una volta la parte più importante del programma:

```
                lea             buffer,a0
                move            #25,d0
                move.b          #'A',d1

        loop    move.b          d1,(a0)+
                addq            #l.dl
                dbra            d0,loop
```

Fig. 6.1: Stampa immediata da A a Z

Che cosa ne pensate di questa alternativa? (Vi garantisco che siamo sempre dentro l'Assembler).

```
for d1 = #'A' to #'Z' do
   move.b d1,(a0)+
endfor
```

oppure:

```
move        #'A',d2
while d2 le #'Z' do
    move.b d2,(a0)+
    addq   #1,d2
endwhile
```

Fig. 6.2: Due soluzioni tramite Macro

La Fig. 6.2 è solo un piccolo estratto dal linguaggio delle macro di un buon Assembler. Se a questo punto vi dico che il codice che ne risulta è esattamente lo stesso come dalla soluzione discreta di cui a Fig. 6.1, sicuramente l'argomento vi interesserà molto..

6.2 Macro

Macro è l'abbreviazione di Macro comando. Macro da solo significa solo grande. In linea di massima una macro è un riassunto di un gruppo di comandi singoli, che abbiamo visto finora, sotto un altro nome. Tali comandi singoli possono venire chiamati anche micro comandi. Purtroppo il linguaggio delle macro non è normalizzato. Ogni Assembler ha la propria sintassi,e quella appena mostrata dell'Assembler GST è già quasi fuori moda. Questo è il motivo per cui tutti gli esempi seguenti sono in linguaggio
macro degli Assembler HiSoft/Metacomco (infatti coincidono) che corrispondono più di ogni altro ai "dialetti macro" della maggior parte degli Assembler. Esempio:

```
CALLEXEC macro
                 move.l   _SysBase,a6
                 jsr       _LVO\1(a6)
                 endm
```

Come già sappiamo questa macro realizza la funzione CALLEXEC (chiamata della funzione della Library Exec). Ogni macro ha un nome che deve trovarsi nel campo Label, seguito dalla parola chiave "macro". Una macro termina con la parola chiave "endm". Fra "macro" e "endm" potrà essere contenuto un quantitativo a piacere di comandi. Se la macro è stata definita, potrà venire richiamata ogni volta che lo si desidera, con il suo proprio nome. All'interno di una macro possono trovarsi anche nomi

di altre macro già precedentemente definite. Consultare tuttavia il manuale per verificare se e per quante volte le macro possono venire inscatolate in questa maniera.

La Fig. 6.3 riporta due macro. esattamente come verrebbero battute all'inizio di un programma.

```
CALLLIB MACRO
        JSR     \1(A6)
        ENDM

LINKLIB MACRO
        MOVE.L \2,A6
        CALLLIB \1
        ENDM
```

Fig. 6.3: Due Macro utilizzate spesso

La macro “CALLLIB” come “Call Library” contiene il già nolo “jsr offset(a6)”.Qui è importante fornire alla macro un parametro, cioè la stringa da inserire. Per tali parametri le macro hanno delle variabili. Per l'Assembler GST (prossimamente disponibile) tali variabili possono essere nomi, tuttavia si incontrano di solito delle cifre con un segno di barra oppure (come nel SEKA) con un punto interrogativo davanti. Nel caso della Metacomco sono permesse anche le lettere da A a Z. Fare tuttavia attenzione che, per la Metacomco e l'HiSoft, la barra deve essere un “Backslash” (barra rovesciata = barra inclinata verso sinistra). Tali limitazioni non sono molto piacevoli (questo simbolo infatti non è presente in tutti gli Editor) ma ormai sono comuni. Per quanto riguarda la seconda macro, sappiamo già che una macro ne può chiamare un'altra.
Quest'ultima però deve essere stata precedentemente definita. Passiamo alla pratica.

ll programma deve far uscire il già noto CIAO. La Fig. 6.4 mostra la soluzione. Paragoniamo questa soluzione con la Fig. 4.1 del Capitolo 4

```
                 opt      l-                         ;non linkare!
_SysBase         equ 4                               ;Base di Exec
_LVOOpenLibrary  equ -552                            ;Apertura della Library
_LVOCloseLibrary equ -414                            ;Chiusura della Library
_LVOOutput       equ -60                             ;DOS: Prelevamento
                          ;dell'Handle di Output
_LVOWrite        equ -48                             ;Output

_main            move.l   #dosname,a1                ;Nome della DOS-Lib
                 moveq    #0,d0                      ;Versione indifferente
                 LINKLIB  OpenLibrary,_SysBase       ;Apertura della DOS-Lib
                 tst.l    D0                         ;Errore?
                 beq      fini                       ;Se errore, Fine
                 move.l   d0,_DOSBase                ;Annotare il puntatore
                 LINKLIB  Output,_DOSBase            ;Prelevamento Handle di
                          ;Output

                 print    d0,#string,20              ;Output del testo
                 move.l   _DOSBase,a1                ;Base della Lib
                 LINKLIB  CloseLibrary,_SysBase      ;Funzione "Chiusura"
fini             rts                                 ;Ritorno al CLI

_DOSBase         dc.l     0
dosname          dc.b     'dos.library',0
                 cnop     0,2
string           dc.b     'Ciao Caro lettore!',10
                 cnop     0,2
```

Fig. 6.4: Un programma con Macro

Questo si presenta già molto bene. Dov'è il problema? Mancano le macro, che vi voglio presentare da sole con la Fig. 6.5.

```
LINKLIB MACRO
        IFNE    NARG-2
        FAIL    ----- Macro LINKLIB: non 2 Argomenti --
        ENDC
        MOVE.L  A6,-(SP)
        MOVE.L  \2,A6
        JSR     _LVO\1(A6)
        MOVE.L  (SP)+,A6
        ENDM
```

```
print   MACRO
        IFNE    NARG-3
        FAIL    -----Macro print: non 3 Argomenti
        ENDC
        MOVE.L  \1,D1             ;Handle di Ouput
        MOVE.L  \2,D2             ;Indirizzo testo
        MOVEQ   #\3.D3            ;Lungh. testo
        LINKLIB Write,_DOSBase    ;Funzione "Scrittura"
        ENDM
```

Fig. 6.5: Le Macro per Fig. 6.4

E' possibile trovare la macro LINKLIB in forme simili nei file Include della Metacomco e della HiSoft, fra gli altri. Le righe

```
        IFNE    NARG-2
        FAIL    ----Macro LINKLIB: Non 2 Argomenti
        ENDC
```

possono anche, in linea di principio, venire tralasciate. Dal momento però che le incontreremo spesso nei file Include (quindi, ne vale la pena) vediamo di spiegarle. NARG è una variabile in Assembler che significa "Number Arguments". Questa variabile è valida solo all'interno di una macro (diversamente è nulla) e mantiene il numero dei parametri con i quali la macro è stata chiamata.

6.2.1 Assemblaggio condizionato

Il fatto che il numero coincida oppure no, è un'altra questione, ma lo si può controllare. Per fare ciò si utilizza una seconda caratteristica di un buon Assembler, cioè l'assemblaggio condizionato. Per esso vale la formula generale

```
        IFcc
        fallo se cc è true
        ENDC
        avanti se cc è false
```

Le condizioni "cc" sono in linea di massima le stesse che abbiamo visto nei comandi di Branch. La limitazione è tuttavia che è possibile confrontare sempre un solo argomento con zero. Quindi volendo controllare se il numero degli argomenti macro (NARG) coincide, ed in caso di errore fare uscire una segnalazione, dovremo dire:

se NARG meno 2 non è uguale a zero

oppure

```
        IFNE NARG-2
```

Non è necessario effettuare questi test a opera d'arte, l'Assembler se ne accorgerà solo più tardi. Se per es. in una macro abbiamo le seguenti righe

```
        print   macro
                MOVE.L \1,D1     ;Handle di Output
                MOVE.L \2,D2     ;Indirizzo testo
```

e chiamiamo questa macro con

```
        print d4
```

risulterà

```
        MOVE    d4,D1
        MOVE      ,D2
```

Alla seconda riga l'Assembler fornirà giustamente una segnalazione,cioè l'errore viene riconosciuto. La soluzione con il "IFcc" potrà mostrare meglio la provenienza dell'errore.

A questo punto avremmo ancora un piccolo problema. Ipotizziamo di aver scritto la seguente macro:

```
        nonsens macro
        loop    move d1,d2
                bra loop
                endm
```

Scriviamo ora nel programma

```
nonsens
nonsens
```

Il processore delle macro svilupperà le seguenti righe:

```
        loop    move d1,d2
                bra loop
        loop    move d1,d2
                bra loop
```

Al più tardi al secondo "bra loop" il povero Assembler comincerà a sbandare. A quale "loop" dovrà saltare? E' chiaro che si fermerà, inviando una segnalazione di errore. Al fine di evitare ciò, nei buoni Assembler c'è sempre una soluzione. Il sistema più efficace lo troviamo per es. nel GST, dove si scriverebbe:

```
nonsens macro
        LOCAL loop
loop    move d1,d2
        bra loop
        endm
```

Con ciò,il "loop" diventa locale, cioè dichiarato solo all'interno delle variabili valide della macro. In altri Assembler troviamo la forma

```
$n invece di "loop"
```

Al posto di n si dovrà inserire un numero tra 1 e 99 (oppure 1 e 999). Tale numero verrà aumentato di uno ad ogni chiamata della macro. Se si utilizzano più macro con Label interne, bisognerà tuttavia fare attenzione a partre con numeri molto diversi fra loro ed eventualmente anche molto distanti. Se torniamo un attimo all'esempio dell'introduzione, cioè

```
for d2 = #A' to #'Z' do
…….
…….
endfor
```

la soluzione dell'indovinello sarà molto semplice. "for", "=", "to" e "endfor" sono delle macro. "do" per es. produrrà solo una label locale, "for" caricherà d2 e "endfor" genererà un "dbra d2,label". Un'altra applicazione sarebbe:

```
ret     Macro
        rts
        endm
```

In questo caso il processore delle macro inserirà, per ogni "ret" che incontrerà in un testo, un "rts". In questa maniera è possibile ridefinire ogni comando dell'Assembler. Ora è anche peggio, vale a dire con la presente:

```
ret     Macro
        dc.b $C9
        endm
```

"ret" nell'Assembler dello Z80 significa Return.

Per il linguaggio macchina dello Z80, un "ret" deve venire tradotto in "$C9". Con ciò sarà possibile scrivere sul nostro Amiga, in Assembler per 68000, un programma in Assembler per lo Z80. Ciò viene chiamato Cross-Assembler e quando possibile, viene effettuato dalle macro. Ora sappiamo anche come programmare un nuovo computer in Assembler, quando per esso non esiste ancora nessun Assembler.

6.2.2 Solo elaborazione dei testi

Nel caso delle macro si tratta in verità di una vera e propria elaborazionedi testi, che con l'Assembler in sé e per sé ha ben poco a che vedere. La Fig. 6.6 riporta un piccolo estratto da un programma, che definisce la macro PRINT e la chiama due volte.

```
PRINT    macro
         movem.l  d0-d3/a6,-(sp)
         jsr      _LVOOutput(a6)
         move.l   d0,d1
         move.l   \1,d2
         move.l   \2,d3
         jsr      _LVOWrite(a6)
         movem.l  (sp)+,d0-d3/a6
         endm

PRINT    #msg1,#len1

PRINT    #msg2,#len2
```

Fig. 6.6: Estratto di programma con Macro

A questo punto ogni buon assemblatore è n grado di produrre il cosiddetto listato Assembler. In Fig. 6.7 è riportato un esempio di ciò,

---

HiSoft GenAmiga Assembler 1.0 page 1

```
4                          PRINT             macro
5                          movem.l  d0-d3/a6,-(sp)
6                          jsr               _LVOOutput(a6)
7                          move.l            d0,d1
8                          move.l            \1,d2
9                          move.l            \2,d3
10                         movem.l  (sp)+,d0-d3/a6
12                         endm
13
14 0000001A +48E7F002      movem.ld0-d3/a6,-(sp)
14 0000001E +4EAEFFC4      jsr_LVOOutput(a6)
14 00000022 +2200move.l    d0,d1
14 00000042 +243C00000072 move.l   #msg1,d2
14 0000002A +263C0000000D move.l   #len1,d3
14 00000030 +4EAEFFD0      jsr               _LVOWrite(a6)
14 00000034 +4CDF400F      movem.l  (sp)+,d0-d3/a6
```

```
15
16 00000038 +48E7F002     movem.ld0-d3/a6,-(sp)
16 0000003C +4EAEFFC4     jsr_LVOOutput(a6)
16 00000040 +2200move.l d0,d1
16 00000042 +243C00000072 move.l   #msg2,d2
16 00000048 +263C00000012 move.l   #len2,d3
16 0000004E +4EAEFFD0     jsr              _LVOWrite(a6)
16 00000052 +4CDF400F     movem.l  (sp)+,d0-d3/a6
17
```

Fig. 6.7: Un listato in Assembler

Nel primo campo, dopo il numero di riga, si trovano gli indirizzi. L'Assembler comincia normalmente con zero. Con ORG è possibile preimpostare un indirizzo di partenza assoluto. L'esempio comincia con 1A, in quanto qui ci troviamo solo di fronte ad una parte. Nel secondo campo ci sono gli “Op-Code” cioè quello che l’Assembler ricava da ogni singolo comando. Si tratta della rappresentazione in esadecimale del linguaggio macchina, che è la sola cosa che il 68000 comprenda. Prima degli Op-code troviamo sempre un segno di +. Non si tratta di un segno, bensì solo di un simbolo indicante che questi comandi provengono da uno sviluppo di macro. A questo punto saremmo arrivati alla parte più interessante. Vediamo chiaramente che le righe 14 e 16 si ripetono (in caso di macro i numeri di riga non vengono incrementati). Sono stati inseriti solo altri valori (msg2 invece di msg1, len2 invece di len1). Ciò significa che , con le macro, i testi in sorgente diventano più corti, il codice oggetto al contrario diventa tanto più lungo, quanto più vengono sviluppate delle macro. Quindi, a partire dalla terza chiamata, vale la pena normalmente utilizzare un sottoprogramma. Un buon anti esempio è quello illustrato in Fig. 6.8.

```
FUNCDEF          MACRO
_LVOl1                   EQU FUNC_CNT
FUNC CNT         SET FUNC CNT-6
                         ENDM
FUNC CNT         SET 4*-6

         FUNCDEF  Supervisor
         FUNCDEF  Exitlntr
         FUNCDEF  Schedule
         FUNCDEF  Reschedule
```

Fig. 6.8: La Macro FUNCDEF

Con questa macro vengono sviluppati gli LVO nel Metacomco. Per I'HiSoft ci si è risparmiati questa deviazione scrivendo immediatamente gli Offset. Al fine di poter capire il senso di questa macro, è necessario chiarire ancora che gli Offset di Library sono sempre negativi. Essi cominciano sempre con -30 ed ogni lnput occupa 6 Byte (il fatto che vengano occupati tutti e 6, e un'altra questione (Risposta: no)). La macro aggiunge sempre -6, per cui si comincia con -24 (4* -6). SET ha lo stesso effetto di EQU, tuttavia con una differenza: con SET si può attribuire un nuovo valore ad una label. EQU è veramente una costante nel senso pieno del termine. Il resto dovrebbe essere comprensibile, se osserviamo in Fig. 6.9 il risultato di Fig. 6.8.

```
FFFFFFE8        _LVOSupervisor   EQU      FUNC_CNT
FFFFFFE2        FUNC_CNT         SET      FUNC_CNT-6
FFFFFFE2        _LVOExitInt      EQU      FUNC_CNT
FFFFFFDC        FUNC_CNT         SET      FUNC_CNT-6
FFFFFFDC        LVOSchedule      EQU      FUNC_CNT
FFFFFFD6        FUNC_CNT         SET      FUNC_CNT-6
FFFFFFD6        _LVOReschedule   EQU      FUNC_CNT
FFFFFFD0        FUNC_CNT         SET      FUNC_CNT-6
```

Fig. 6.9: Il risultato del programma di Fig. 6.8

Per i numeri, è necessario osservare sempre la notazione in complemento di due. Gli FFFFFF che stanno davanti possono venire, per il momento, tralasciati. Il rimanente E2 (esa) per es. è 226 in decimale. 226 - 256 dà come risultato -30. Fare attenzione che il supervisore _LVO viene dapprima impostato con EQU a -24 (E8). Solo l'istruzione SET successiva modificherà il valore a -30 (E2).

6.3 file “Include”

Anche le istruzioni di Include possono venire sostituite con delle macro. Ipotizziamo di avere memorizzato la macro definizione di Fig. 6.6 in un file di testo con il nome “Mac66”. Sarà quindi possibile scrivere il programma di cui alla Fig. 6.6 come mostrato in Fig. 6.10.

```
                include "Mac66

                PRINT #msg1,#len1
                PRINT #msg2,#len2


msg1            dc.b  ‘Ciao!’
len1            equ *-msg1
                ds.w 0
msg2            dc.b ‘Amiga’
len2            equ *-msg2
                end
```

Fig. 6.10: programma con Macro che vengono lette da un file di Include

Questo metodo è molto utile, in quanto inseriremo sicuramente moltissime funzioni DOS in ogni programma. Se definiamo tali funzioni una volta per tutte come macro e le memorizziamo nel “Mac-file” ci risparmiamo non solo lunghe sequenze di battitura, ma anche un notevole tempo per la ricerca di errori, grazie alla mancanza di errori di battitura. D’altra parte, non fa nulla se in un programma molte macro non vengono utilizzate. In tal caso non produrranno nessun codice. Se la “Mac-Lib” (abbreviazione per Library = Biblioteca) diventa troppo grande, occorrerà naturalmente molto tempo quando l'Assembler la dovrà leggere ad ogni passaggio. In caso di un RAM-Disk, ciò potrà portare a problemi di spazio.

Questo è il motivo per cui la “Lib” dovrebbe venire suddivisa in molti piccoli file, a seconda degli argomenti. Lasciamo per il momento in sospeso la domanda, se sia il caso o no di portare questo fatto alle sue estreme conseguenze, come nei file Include della Metacomco e HiSoft. Ci sono per es. delle macro che definiscono i nomi delle Library. In questo modo non sarà necessario annotare tali nomi, bensì quelli delle macro. La macro seguente è assolutamente superflua, a meno che qualcuno non voglia aprire la Exec-Lib.

```
EXECNAME macro
         dc.b     'exec.library',0
         even
         endm
```

Inoltre è veramente consigliabile utilizzare i file Include dei sistemi Assembler. Osserviamo tali file con tranquillità con un Editor. Tramite essi si potrà imparare molto sulle strutture dei dati dell'Amiga (affronteremo più approfonditamente questo argomento in seguito)

6.4 Moduli

I moduli forniscono un aiuto ulteriore per la razionalizzazione del lavoro di programmazione. La filosofia che si cela dietro di essi, è quella di suddividere un programma lungo in molti blocchi singoli, possibilmente indipendenti l'uno dall'altro, oppure sezioni, chiamate anche brevemente moduli. Un linguaggio come il Modula, per es., è nato da tale filosofia. In Assembler bisogna distinguere tra due tipi di moduli, cioè
- moduli di testo
- moduli di codice

6.4.1 Moduli di testo

Abbiamo già praticamente incontrato i moduli di testo, in quanto si tratta dei file include. Un programma modularizzato a livello di testo, con il titolo di lavoro DED, potrebbe essere come segue:

```
include “dos.mac”
include “bios.mac”
Include “ded_logo”
include “ded_menu”
include “ded subs”
```

Il vantaggio del modulo di testo si trova principalmente nel fatto che i listati diventano relativamente corti, se si memorizza sempre come modulo di testo una parte pronta.
Se per es. nello sviluppo di una parte mi trovo a “ded_menu”, nel quale naturalmente c’è un errore, non ho bisogno di elaborare tutto fino alla riga 447. ma arriverò immediatamente alla riga 5.

Il secondo vantaggio sarebbe naturalmente che si possono riutilizzare tutte le parti utilizzabili in generale, come per es. le “DOS-Lib”.

6.4.2 Moduli Code

Uno svantaggio del modulo di testo è che deve venire riassemblato ad ogni passaggio. Un aiuto per fare ciò viene offerto dai moduli Code. Infatti singoli blocchi vengono assemblati separatamente e dopo ciò collegati al programma principale dal Linker. E’ molto comodo, ma ha bisogno di un po' più di lavoro, e presenta determinate esigenze di Linker e dell'intero ambiente di programmazione. Diciamo quindi subito che vale la pena di utilizzarli solo in caso di programmi lunghi e lunghissimi.
Cominciamo con il superlavoro del programma. In un programma completo potremmo dire per es. quanto segue:

```
bsr print.
```

Se però la routine di Print si trova in un altro modulo, sarà necessario comunicare al programma che “print” è una routine esterna. Lo stesso vale per i nomi delle variabili. Per questi motivi l'Assembler dovrà offrire delle direttive del tipo

```
        External,
        Global
e/o     XREF
```

Tali direttive devono naturalmente venire utilizzate anche dal lettore. A seconda dell'Assembler, le si potrà usare di più o di meno, naturalmente anche ciò non è stato standardizzato, per cui sarà necessario che il lettore analizzi la tematica e sia in grado di dominare l'argomento.

Se non ci siamo spaventati ed abbiamo già modularizzato il nostro programma, incontriamo il problema successivo.

Premettiamo che il Linker può collegare un numero di moduli a piacere (attenzione, alcuni permettono solo delle righe di Input per es. di 64 caratteri!). Quindi sarà sempre un lavoro inutile chiamare il Linker tutte le volte, ci si presenta infatti una delle due soluzioni seguenti:
1. L’intero passaggio viene elaborato “in Batch” (Ved. Cap. 4).
2. Il Linker offre una istruzione come “INPUT File-Name”. In questo caso scriveremo, una volta per tutte, tutte le istruzioni di Linker in un file di testo. Alla chiamata del Linker gli diremo quindi di utilizzare tale file di testo.

Certo, la cosa è un po' laboriosa. Posso solo consigliarvi di riazzerare dapprima il modulo Code dell’argomento. Solo quando saremo in grado di dominare completamente l'Assembler per il 68000; e quando ci dedicheremo a grossi compiti, come per es. lo sviluppo di un interprete BASIC, avremo bisogno di approfondire l'argomento modularizzazione a livello di codice (e in tal caso sarà veramente urgente).

D'altra parte, la programmazione modulare in Assembler è l'unico sistema per “cavarne i piedi” anche in caso di programmi medio-grandi. Vediamo nell'esempio seguente come si procede in pratica.

Sarà necessario sviluppare un “Diskeditor”. Nel menu principale l'utente si trova di fronte alla scelta fra i comandi “L(lettura), E(editaggio), S(scrittura) e Exit”. Lasciamo perdere per il momento, come il menu viene offerto. Fermiamoci al fatto che l'immissione delle lettere L, E, S e X devono far partire l'azione corrispondente. Possiamo quindi dire che ci troviamo di fronte ad un file Include, che legge un tasto e chiama il sotto programma ad esso attribuito. Tale file viene mostrato in Fig. 6.11. Si tratta di una parte del programma “CASE X OF” del Capitolo 5.

```
* Start.icl

start
* lettura tasto dopo d0……
        bclr     #5,d0              ;Maiuscolo forzato
        lea      keys,a0            ;Tabella tasti validi
        move     #count,d1          ;Loro numero
search  cmp.b    (a0)+,d0           ;Tasto su posizione attuale?
        dbeq     d1,search          ;se no, cercare ancora
                                    ;fino a fine tabella
        tst      d1                 ;Tasto trovato?
        bmi      start              ;se no, a uno nuovo
        neg      d1                 ;sub d1, #count
        add      #count,d1          ;fornisce il nr. di posiz. del tasto
        lsl      #2,d1              ;che per 4
        lea      table,a0           ;Indir. della Routine
        move.l   0(a0,d1.w),a0      ;da determinare
        jsr      (a0)               ;chiamata della Routine
        bra      start
```

Fig. 6.11: Modulo di partenza sotto forma di file include

Dato che questo modulo esiste, cominciamo ora con il programma nuovo, come mostrato in Fig. 6.12.

```
        include "start.icl"
Lett.   rts
Edit    rts
Scritt. rts
keys    dc.b     'L','E','S','X'
count   equ      *-keys
        ds.w      0
table   dc.l     Lett, Edit, Scritt., Exit
```

Fig. 6.12: Un nuovo programma comincerà così

6.5 Top Down Bottom Up

Vediamo immediatamente la struttura del programma. Dapprima non ci interessa sapere che cosa accade nei singoli sottoprogrammi. Procediamo e riempiamo i singoli sottoprogrammi uno dopo l'altro. Quando un sottoprogramma è pronto, esso verrà verificato. Solo quando gira, potrà cominciare quello successivo. Praticamente si va avanti di un passo. Per es. il sottoprogramma "leggi" ha bisogno di una routine che legga un settore e di un'altra che faccia uscire sullo schermo in esadecimale il settore letto.
Per fare ciò ho bisogno, fra l'altro,di un sottoprogramma "visualizza". Visualizza però ha bisogno di una routine, che trasformi una parola nella stringa ASCII corrispondente.
Da ciò avremo il seguente percorso:

```
Lett.           bsr     read_sec
                bsr     visual.
                rts

read_sec        rts

visual.         bsr     trasf.
                bsr     print
                rts

trasf.          rts
print           rts
```

Ho cominciato dall'alto e sono arrivato, infine, al sottoprogramma "print". Tale sottoprogramma deve venire ora veramente elaborato. Quando la routine "print" gira, posso far partire "converti". Infatti è solo a questo punto che posso far uscire le cifre esadecimali convertite, e di conseguenza testare la routine "print". Ora devo eseguire "visualizza", la quale, dopo ripetute chiamate di print, fa uscire sullo schermo il contenuto di un Buffer. Dopo ciò scriverò "read_sec" cosa che riempirà il Buffer di dati.
Solo ora posso chiamare nel menu principale "leggi" e a questo punto ci troviamo di nuovo in cima.

Questo procedimento "dall'alto verso il basso e ritorno" viene chiamato "top down bottom up". Si tratta di un metodo di programmazione consigliabile anche in Assembler. Con ciò si limita la ricerca di errori sempre solo ad un campo molto piccolo e quindi osservabile. Non evitiamo quindi quel po' di superlavoro necessario a fornire temporaneamente i dati ai singoli sottoprogrammi. Tale superlavoro non è molto. mentre il vantaggio è enorme. Per es. la routine "converti" dovrà far uscire una parola in D0 sotto forma di stringa esadecimale. Scriviamo quindi semplicemente

```
move $19AF,d0
```

come prima riga nel sottoprogramma “converti”. Se ora verifichiamo la routine e vediamo sullo schermo “19AF”, possiamo essere assolutamente sicuri che “converti” funziona.

CAPITOLO 7

Sviluppo dei programmi passo passo

Esempio di “bindec”

7.1 Il principio della conversione di numeri binari in stringhe

Nel presente capitolo mostreremo come si sviluppa un programma passo passo. Come esempio utile serve una routine di cui in seguito avremo molto bisogno. Il suo nome è “bindec”. “bindec” deve convertire un valore intero positivo (0...65535), come lo vede il 68000, cioè binario, in una stringa decimale, che possa venire letta da noi.

La Fig. 7.1 mostra il primo passo da cui cominciare a studiare la tecnica di base della conversione dei numeri.

Premettiamo che per il primo utilizzo è più comodo fare uscire immediatamente ogni posizione (ogni numero) e non raccogliere dapprima tutti i caratteri in un Buffer. Inoltre esistono molti listati, destinati ai computer che lavorano orientati ai caratteri, che mettono a disposizione routine con nomi come CONOUT (Esempio: AtariST). Di conseguenza, e al fine di poter comprendere più semplicemente anche i listati “forestieri” simuliamo un CONOUT tramite una macro con lo stesso nome. CONOUT corrisponde alla macro PRINT di cui al Capitolo 6, con la sola differenza che qui la lunghezza è 1 costante, e di conseguenza non deve mai venire trasferita.

```
                 Opt                l-
* dec1
                 include   OpenDos.i

CONOUT           macro
                 movem.l   d0-d3/a6,-(sp)
                 jsr       _LVOOutput(a6)    ;Prelevamento dell'handle di
                  ;output
                 move.l    d0,d1             ;Handle di output
                 move.l    \1,d2             ;Indirizzo del testo
                 move.l    #1,d3             ;Lunghezza testo
                 jsr       _LVOWrite(a6)     ;Funzione "Scrittura"
                 movem.l   (sp)+,d0-d3/a6
                 endm

                 move      #62345,d2         ;Numero del testo
                 and.l     #$FFFF,d2         ;Delimitazione a Parola
                 divs      #10000,d2         ;posizione 10000
                 bsr       out               ;output

                 swap      d2                ;Resto divisione per d2.w
                 and.l     #$FFFF,d2         ;di nuovo dimensionamento a
                  ;Parola
                 divs      #1000,d2          ;ora la posizione 1000
                 bsr                out
```

```
                swap    d2              ;come sopra per la pos. 100
                and.l   #$FFFF,d2
                divs    #100,d2
                bsr     out

                swap    d2              ;ora la pos. 10
                and.l   #$FFFF,d2
                divs    #10,d2
                bsr     out

                swap    d2              ;e la 1
                bsr     out

                move.b  #10,buffer
                CONOUT  #buffer

                move.l  a6,a1           ;Base di DOS-Lib
                move.l  _SysBase,a6     ;Base di Exec
                jsr     _LVOCloseLibrary(a6)     ;Funzione "Chiusura"

fini            rts                     ;Ritorno al CLI


out             add.b   #'0',d2         ;trasformazione in ASCII
                move.b  d2,buffer
                CONOUT  #buffer         ;Emissione di 1 carattere
                rts

* Campo dati:

dosname         dc.b    'dos.library',0
                cnop    0,2

buffer          ds.b    80
```

Fig 7.1: Il principio di "bindec"

Il principio della conversione dei numeri è molto semplice. Dobbiamo esprimere il numero, per es. 123, come forma di 1 centinaia, 2 decine e 3 unita. Dopo che avremo così isolato le cifre 1, 2 e 3, per il computer sono sempre solo numeri, ma a questo punto bisogna aggiungere ancora il codice ASCII del simbolo "0" per ottenere dei caratteri stampabili. Il metodo dell'isolamento delle singole cifre è la divisione continua, cioè:

```
123 /100 = 1 resto 23
 23 /  10 = 2 resto  3
  3 /   1 = 3 resto  0
```

La divisione viene effettuata dal 68000 tramite il comando

```
DIVS oppure DIVU
```

Ciò significa "Division Signed" (con segno) oppure "Division Unsigned" (senza segno). Un dividendo a 32 Bit viene diviso per un divisore a 16 Bit. Si divide sempre destinazione/sorgente. Di conseguenza il risultato si trova negli operandi di destinazione, cioè

Parola più elevata	Parola più bassa
Resto	Quoziente

Il programma di cui alla Fig. 7.1 dovrà convertire la parola che si trova nel Registro D2 in decimale. Dal momento che possiamo operare solo con parole, un eventuale dividendo in parola lunga dovrà venire limitato alla lunghezza di una parola (in D2) tramite il comando "and.l". Ora D2 viene diviso per 10.000. Il risultato e il valore della posizione decimillesima, che viene fatto uscire nel sottoprogramma "out". Ora, tramite il comando SWAP, il resto viene portato nella parola a valore più basso. La quale verrà di nuovo limitata "a parola" e quindi divisa per 1000. Il processo continua per la posizione delle centinaia e delle decine. Alla posizione delle unità non dovremo naturalmente più dividere, ma non dobbiamo nemmeno dimenticarla.

In questo modo abbiamo evitato molte ripetizioni, cioè, abbiamo razionalizzato. La prima applicazione è mostrata in Fig. 7.2.

```
        opt     l-
* dec2
        include OpenDos.i
        include conout.i

        move    #12345,d2                ;Numero del testo

        move    #10000,d1                ;Posizione 10000
        bsr     out2                     ;output

        move    #1000,d1         ;lo stesso per posizione 1000
        bsr     out1

        move    #100,d1
        bsr     out1

        move    #10,d1
        bsr     out1
```

```
        move    #1,d1
        bsr     out1

        move.b  #10,buffer
        CONOUT  #buffer

        move.l  a6,a1                   ;Base di DOS-Lib
        move.l  _SysBase,a6             ;Base di Exec
        jsr     _LVOCloseLibrary(a6)    ;Funzione "Chiusura"

fini    rts                             ;Ritorno al CLI


out1    swap    d2                      ;Resto divisione per d2.w
out2    and.l   #$FFFF,d2               ;di nuovo  a Parola
        divs    d1,d2                   ;Prelevamento posizione
        add.b   #'0',d2                 ;trasformazione in ASCII
        move.b  d2,buffer
        CONOUT  #buffer                 ;Emissione di 1 carattere
        rts

* Campo dati:

dosname dc.b    'dos.library',0
        cnop    0,2

buffer  ds.b    80
```

Fig. 7.2: Prima razionalizzazione: più lavoro nel sottoprogramma

Torniamo indietro di un passo: la macro CONOUT è stata memorizzata nel frattempo nel file Include “conout.i”. Selezioniamo tale blocco dalla Fig. 7.1 e memorizziamolo su disco sotto il nome “conout.i”.

Vediamo quindi che i comandi “swap”, “ext.l” e “divs” si sono trasformati nel sottoprogramma. Dal momento che alla prima chiamata non è tuttavia possibile “scambiare” (swap), il sottoprogramma è stato dotato di due punti di ingresso. Si tratta di un trucco molto amato, ma anche molto particolare. Il divisore viene trasferito in ogni caso nel Registro D1. Ora la cosa che disturba è che esistono cinque chiamate di sottoprogramma quasi uguali fra loro. Come modificare ciò è mostrato in Fig. 7.3.

```
        opt      l-
* dec3
        include  OpenDos.i
        include  conout.i

        move     #12345,d2                ;Numero del testo

        move.l   #10000,d1                ;Posizione 10000. Ora e' Lunga!
        bsr      out2                     ;output

        move     #3,d3
loop    divs     #10,d1
        bsr      out1
        dbra     d3,loop


        move.b   #10,buffer
        CONOUT   #buffer

        move.l   a6,a1                    ;Base di DOS-Lib
        move.l   _SysBase,a6              ;Base di Exec
        jsr      _LVOCloseLibrary(a6)     ;Funzione "Chiusura"

fini    rts                               ;Ritorno al CLI


out1    swap     d2                       ;Resto Divisione per d2.w
out2    and.l    #$FFFF,d2                ;di nuovo a Parola
        divs     d1,d2                    ;Prelevamento posizione
        add.b    #'0',d2                  ;Trasformazione ASCII
        move.b   d2,buffer
        CONOUT   #buffer                  ;Emissione di 1 carattere
        rts

* Campo dati:

dosname dc.b     'dos.library',0
        cnop     0,2

buffer  ds.b     80
```

Fig. 7.3: Seconda razionalizzazione: aggiunta di un loop

Il Loop è stato costituito tramite il “dbra” già noto, dove D3 serve come contatore. All’interno del Loop lo stesso divisore D1 viene diviso sempre per 10. Fare comunque attenzione che D1 viene qui inizializzato con una costante lunga. Dal momento che il Loop deve girare fino alle unità, quest’ultima verrà infine divisa inutilmente per 1. Evitare ciò, tuttavia, farà perdere ulteriore tempo, quindi lasciamo tutto com’è. Ho solo

detto che il sottoprogramma è ridondante. La Fig. 7.4 mostra come portare il sottoprogramma nel Loop.

```
        opt     l-
* dec4
        include OpenDos.i
        include conout.i

        move    #123,d2                     ;Numero del testo

        move.l  #10000,d1                   ;Posizione 10000. Ora e' lunga!
        move    #4,d3                       ;il contatore di loop e' ora a 4
        bra     out2                        ;output

loop    divs    #10,d1
out1    swap    d2                          ;Resto Divisione per d2.w
out2    and.l   #$FFFF,d2                   ;di nuovo a Parola
        divs    d1,d2                       ;Prelevamento posizione
        add.b   #'0',d2                     ;trasformazione in ASCII
        move.b  d2,buffer
        CONOUT  #buffer                     ;Emissione di 1 carattere
        dbra    d3,loop

        move.b  #10,buffer
        CONOUT  #buffer

        move.l  a6,a1                       ;Base di DOS-Lib
        move.l  _SysBase,a6                 ;Base di Exec
        jsr     _LVOCloseLibrary(a6)        ;Funzione "Chiusura"

fini    rts                                 ;Ritorno al CLI


* Campo dati:

dosname dc.b    'dos.library',0
        cnop    0,2

buffer  ds.b    80
```

Fig. 7.4: Terza razionalizzazione: abbandono dei sottoprogrammi

```
        opt      l-
* dec5
        include  OpenDos.i
        include  conout.i

        move     #123,d2                    ;Numero di testo

        clr      d4                         ;Soppressione del flag Zero

        move.l   #10000,d1                  ;Posizione 10000. Ora e' lunga!
        move     #4,d3                      ;il contatore di loop e' ora a 4
        bra      out2                       ;output

loop    divs     #10,d1
out1    swap     d2                         ;Resto Divisione per d2.w
out2    and.l    #$FFFF,d2                  ;di nuovo a Parola
        divs     d1,d2                      ;Prelevamento posizione
        add.b    #'0',d2                    ;trasformazione in ASCI

        cmp.b    #'0',d2                    ;e' uno Zero?
        bne      out3                       ;se no, output
        tst      d4                         ;Spazi bianchi permessi?
        bne      out3                       ;no: output di zero
        move.b   #' ',d2                    ;si  : imposta spazio bianco
        bra      out4                       ;      quindi output
out3    move     #1,d4                      ;Flag di fine spazi bianchi

out4    move.b   d2,buffer
        CONOUT   #buffer                    ;Emissione di 1 carattere
        dbra     d3,loop

        move.b   #10,buffer
        CONOUT   #buffer

        move.l   a6,a1                      ;Base di DOS-Lib
        move.l   _SysBase,a6                ;Base di Exec
        jsr      _LVOCloseLibrary(a6)       ;Funzione "Chiusura"

fini    rts                                 ;Ritorno al CLI


* Campo dati:

dosname dc.b     'dos.library',0
        cnop     0,2

buffer  ds.b     80
```

Fig. 7.5: Quarta razionalizzazione: soppressione degli zeri iniziali

Fare comunque attenzione al fatto che, a causa dello “swap”, che all'inizio volevamo evitare ci troviamo ora nel Loop per cui dobbiamo inizializzare il contatore del Loop con 4. Sospendiamo per il momento la logica e occupiamoci dell'estetica. Avrete sicuramente notato che il programma scrive degli zeri all'inizio, quando i numeri sono più corti di cinque posizioni. Sarà possibile eliminare tali zeri facendo uscire degli spazi bianchi al loro posto. Come fare ciò è mostrato in Fig. 7.5.

Il problema è semplice da descrivere. Gli zeri iniziali devono venire sostituiti da spazi bianchi, gli altri zeri no. Per fare ciò è necessario un Flag (marcatore), che venga impostato tutte le volte che appare un numero diverso da zero. Questo per quanto riguarda la logica. Tuttavia è più pratico verificare leggermente il procedimento logico: ogni numero diverso da zero imposta il Flag. Con ciò ci si risparmia la comprensione, abbastanza complessa da realizzare, della differenza del primo “non zero” e degli altri zeri.

Nel nostro caso il Registro D4 costituisce il Flag. L'interrogazione comincia nella riga con “+ + + nuovo + + +” al bordo destro. Se il numero non è zero, D4 viene caricato con 1, quindi il numero viene fatto uscire. Diversamente, dovrà essere uno zero. Ora viene il test. Se D4 è impostato, lo zero viene stampato come zero. Diversamente, D2 viene caricato con uno spazio bianco.

Avevamo deciso che questo deve essere un sottoprogramma utilizzabile universalmente. Ciò significa dapprima che il sottoprogramma non potrà far uscire i caratteri sullo schermo, perché altrimenti non si potrà, per es. stampare. La modifica non è un problema. La Fig. 7.6 mostra la soluzione. L’emissione ha luogo in un Buffer. Il Registro A0 funziona come puntatore di Buffer al fine di sapere quanti caratteri sono validi nel Buffer, viene scritto come ultimo carattere un Byte di 0. Ciò è molto pratico, e con ciò abbiamo ottenuto contemporaneamente una stringa DOS.

```
        opt      l-
* dec6
        include  OpenDos.i
        include  conout.i

PRINT   macro
        movem.l  d0-d3/a6,-(sp)
        jsr      _LVOOutput(a6)           ;Prelevamento dell'Handle di
output
        move.l   d0,d1                    ;Handle di output
        move.l   \1,d2                    ;Indirizzo del testo
        move.l   \2,d3                    ;Lunghezza testo
        jsr      _LVOWrite(a6)            ;Funzione "Scrittura"
        movem.l  (sp)+,d0-d3/a6
        endm


        move     #123,d2                   ;Numero del testo
```

```
        clr     d4                         ;Soppressione del Flag di zero
        lea     buffer,a0                  ;Buffer di risultato

        move.l  #10000,d1                  ;Posizione 10000. Ora e' lunga!
        move    #4,d3                      ;il contatore di loop e' ora a 4!
        bra     out2                       ;output


loop    divs    #10,d1
out1    swap    d2                         ;Resto Divisione perd2.w
out2    and.l   #$FFFF,d2                  ;di nuovo a parola
        divs    d1,d2                      ;Prelevamento posizione
        add.b   #'0',d2                    ;trasformazione in ASCII

        cmp.b   #'0',d2                    ;e' uno Zero?
        bne     out3                       ;se no, output
        tst     d4                         ;Spazi bianchi permessi?
        Bne     out3                       ;no: emissione di zero
        move.b  #' ',d2                    ;si  : impostazione spazio bianco
        bra     out4                       ;      quindi output
out3    move    #1,d4                      ;Flag di fine spazi bianchi
out4   move.b   d2,(a0)+                   ;Carattere -> Buffer
        dbra    d3,loop
        move.b  #0,(a0)                    ;ignorato da PRINT
        PRINT   #buffer,#5
        move.w  #$0A0A,buffer              ;2 Linefeeds
        PRINT   #buffer,#2

        move.l  a6,a1                      ;Base di DOS-Lib
        move.l  _SysBase,a6                ;Base di Exec
        jsr     _LVOCloseLibrary(a6)       ;Funzione "Chiusura"

fini    rts                                ;Ritorno al CLI


* Campo dati:

dosname dc.b    'dos.library',0
        cnop    0,2

buffer  ds.b    80
```

Fig. 7.6: Quinta razionalizzazione: Output in una stringa

Tuttavia non si tratta ancora di un sottoprogramma veramente universale. E' già sgradevole il fatto che il trasferimento dei valori ha luogo nel Registro D2, cosa che potrebbe venire utilizzata dal programma principale, ancora peggio è il fatto che a questo programma è abbinata stabilmente una variabile chiamata “Buffer”. Ciò deve venire modificato, come mostrato in Fig. 7.7.

```
        opt      l-
* dec7
        include  OpenDos.i
        include  conout.i

PRINT   macro
        movem.l  d0-d3/a6,-(sp)
        jsr      _LVOOutput(a6)    ;Prelevamento dell'Handle di Output
        move.l   d0,d1             ;Handle di output
        move.l   \1,d2             ;Indirizzo testo
        move.l   \2,d3             ;Lunghezza testo
        jsr      _LVOWrite(a6)     ;Funzione  "Scrittura"
        movem.l  (sp)+,d0-d3/a6
        endm


        move     #1001,-(sp)       ;Numero testo
        pea      buffer            ;Buffer risultato
        bsr      bindec            ;Chiamata della Routine

        PRINT    #buffer,#5        ;Emissione del numero

        move.w   #$0A0A,buffer     ;2 Linefeeds
        PRINT    #buffer,#2


        move.l   a6,a1             ;Base di DOS-Lib
        move.l   _SysBase,a6       ;Base di Exec
        jsr      _LVOCloseLibrary(a6)      ;Funzione "Chiusura"

fini    rts                        ;Ritorno al CLI

;------------------------------------------------------------------
bindec  move.l   4(sp),a0          ;Prelevamento indirizzo del buffer
        move     8(sp),d2          ;e del numero da trasformare
        move.l   #10000,d1         ;primo Divisore
        move     #4,d3             ;Contatore di loop
        clr      d4                ;soppressione Flag di zero
        lea      buffer,a0         ;Buffer di risultato
        bra      out2              ;Trasformazione in 10000
loop    divs     #10,d1            ;e da 1000 a 1
out1    swap     d2                ;Resto della Divisione dopo d2.w
out2    and.l    #$FFFF,d2         ;di nuovo a Parola
        divs     d1,d2             ;prossima posizione
        add.b    #'0',d2           ;Trasformazione in ASCII

        cmpi.b   #'0',d2           ;e' uno Zero?
        Bne      out3              ;se no , output
        tst      d4                ;Spazi bianchi permessi?
        Bne      out3              ;no, output di Zeri
        move     #' ',d2           ;impostazione spazi bianchi
        bra      out4              ;e output
out3    move     #1,d4             ;Flag di fine spazi bianchi
```

```
out4    move.b  d2,(a0)+          ;Carattere -> Buffer
        dbra    d3,loop
        move.b  #0,(a0)           ;Carattere di fine

        move.l  (sp)+,a0          ;Prelevamento indirizzo di Return
        addq.l  #6,sp             ;Parametri dello Stack
        jmp     (a0)              ;e return


* Campo dati:

dosname dc.b    'dos.library',0
        cnop    0,2

buffer  ds.b    80
```

Fig. 7.7: Sesta razionalizzazione: trasferimento parametri tramite lo Stack

Il sottoprogramma “bindec” viene chiamato nella seguente sequenza
- Valore sullo Stack
- Indirizzo di Buffer sullo Stack
- `bsr      bindec`

Dopo ciò, sullo Stack avremo
8(sp): Valore
4(sp): Indirizzo di Buffer
0(sp): Indirizzo di Return

Di conseguenza ci si può procurare tali parametri con:

```
        move.l          4(sp),a0
        move.w          8(sp),d2
```

Il resto funziona come noto, tuttavia con una piccola differenza alla fine.

```
        move.l   (sp)+,a0
```

prende l'indirizzo di Return dallo Stack ed incrementa il puntatore allo Stack di 4. Ora dovremo solo “eliminare” i 6 Byte dei parametri (parola per valore e parola lunga per indirizzo) dallo Stack. Ciò accade tramite l'istruzione

```
        addq.l  #6,sp
```

Alla fine si dovrà naturalmente avere un “Return” che però in questo caso significa solo “salta all’indirizzo Return”, quindi

```
        jmp     (a0)
```

Probabilmente questo meccanismo non giunge nuovo, in quanto i linguaggi evoluti funzionano in modo simile. E’ molto interessante sapere che il linguaggio C, che vuole essere il linguaggio ideale per l’Amiga, si occupa purtroppo esclusivamente del proprio trasferimento di parametri, il quale, come noto, gira su registri. Infine, ogni funzione del C passa tutti i parametri allo Stack. Dopo ciò, il compilatore aggiunge una routine che andrà a riprendere i parametri dallo Stack e li caricherà nel Registro. E' facile immaginare che questa deviazione costerà tempo e codici, ciò nonostante é necessario studiare bene questo meccanismo, che bisogna assolutamente conoscere per potersi occupare del linkaggio di routine Assembler in BASIC. Come mostrato nel relativo capitolo.

```
bindec  move.l   4(sp),a0      ;Prelevamento indirizzo del buffer
        move     8(sp),d2      ;e del numero da trasformare
        move.l   #10000,d1     ;primo Divisore
        move     #4,d3         ;Contatore di loop
        clr      d4            ;soppressione Flag di zero
        lea      buffer,a0     ;Buffere di risultato
        bra      out2          ;Trasformazione in 10000
loop    divs     #10,d1        ;e da 1000 a 1
out1    swap     d2            ;Resto della Divisione dopo d2.w
out2    and.l    #$FFFF,d2     ;di nuovo a Parola
        divs     d1,d2         ;prossima posizione
        add.b    #'0',d2       ;Trasformazione in ASCII

        cmpi.b   #'0',d2       ;e' uno Zero?
        Bne      out3          ;se no , output
        tst      d4            ;Spazi bianchi permessi?
        bne      out3          ;no, output di Zeri
        move     #' ',d2       ;impostazione spazi bianchi
        bra      out4          ;e output
out3    move     #1,d4         ;Flag di fine spazi bianchi

out4    move.b   d2,(a0)+      ;Carattere -> Buffer
        dbra     d3,loop
        move.b   #0,(a0)       ;Carattere di fine

        move.l   (sp)+,a0      ;Prelevamento indirizzo di Return
        addq.l   #6,sp         ;Parametri dello Stack
        jmp      (a0)          ;e return
```

Fig. 7.8: Ultima razionalizzazione: i registri di lavoro vengono assicurati

Siamo a buon punto, ma non siamo ancora perfetti. Il nostro “bindec” disturba purtroppo i Registri da D1 a D4. Anche A0 viene modificato, ma ciò è usuale, ed anche D0 è sempre “scratch” (di lavoro).

Ho modificato le label, per cui non ne troveremo di simili in altri programmi. Nel caso in cui l'Assembler di cui si dispone offra delle label locali, è il caso di utilizzarle (per il Metacomco $n...). Il comando

```
        movem.l  d1-d4,-(sp)
```

è nuovo. Tramite esso è possibile portare un intero gruppo o lista di registri nello Stack. E’ permesso scrivere anche:

```
        movem.l  d1-d4/a1-a2/a5,-(sp)
```

Il contrario (prelevamento dallo Stack) sarà quindi:

```
        movem.l  (sp)+,d1-d4/a1-a2/a5
```

Dal momento che nel nostro esempio sullo Stack si trovano solo quattro registri, cioè 16 Byte, al fine di poter prendere i parametri, dovremo fornire anche questi 16 Byte:

```
        move.l   20(sp),a0        ;Prelevamento indirizzo Buffer
        move     24(sp),d2        ;e numero da trasformare
```

Ora memorizziamo la routine “bindec” in un file extra, in quanto ne avremo bisogno in seguito.

CAPITOLO 8

Breve corso sull'Intuition

8.1 Multitasking

Se si fa girare sotto CLI il proprio programma, esso verrà trattato come un sotto-programma del CLI. Di conseguenza esso può venire scritto senza codice di Startup e terminare semplicemente con RTS. Se al contrario un programma deve girare come Task, esso dovrà venire dotato di un codice di Startup (che vedremo meglio in Capitolo 9). Il significato di codice di Startup è fra l'altro, che un programma dovrà attendere un messaggio prima di poter partire e con ciò siamo entrati nell'argomento Multitasking.

Il nostro Task (programma) può comportarsi in linea di principio come se fosse l'unico Task del sistema in pratica però, non si va molto avanti, in quanto è necessario comunicare con altri Task, per es. al fine di apprendere se è stato mosso il Mouse. Nel caso dell'Amiga, la comunicazione ha luogo tramite porte di messaggi. Tutto ciò viene gestito dalla routine di smistamento dei messaggi, che potremmo paragonare all'impiegata del centralino, che rende possibile i collegamenti fra i singoli abbonati. Nel nostro caso, però, abbiamo una supertelefonista. Se il nostro numero per es, è occupato, la ragazza prenderà un appunto e ci fornirà la notizia solo quando saremo di nuovo liberi. Usando un linguaggio per adepti diremo che le notizie vengono bufferizzate in una “Message-Queue” (coda dei messaggi). Naturalmente sarà necessario per lo meno possedere un telefono, cioè una porta per messaggi per il Task.

La cosa tipica di un Task, è che esso aspetta un messaggio, cioè che l'utente prema un tasto oppure sposti il Mouse. Ciò significa che tutti gli Input girano su di un Task sotto il nome “input.device”. L'Amiga si occupa solo del fatto che la notizia pervenga ai Task (finestra) e noi dobbiamo attendere che ciò accada. Il metodo più grossolano è il cosiddetto “Polling”. Con esso, il Task va in un Loop, dal quale egli interroga la porta messaggi finché non trova un messaggio. Con ciò, però, esso utilizza tempo di calcolo, che mancherà agli altri Task. E' molto meglio utilizzare una funzione chiamata Wait. In questo caso il Task viene cancellato dalla lista dei Task attivi e viene messo nella lista dei Task in attesa, da dove egli non potrà impedire il funzionamento degli altri. Si dice anche che il Task “dorme”. Il sistema operativo, al contrario, resta sveglio e controlla continuamente se arriva una notizia per tale Task. Se ciò accade il Task viene di nuovo svegliato (prosegue dopo la chiamata di Wait) e potrà quindi gestire il messaggio, per es. reagire alla pressione di un tasto, come previsto dal programma.

La forma più semplice di Wait è la funzione WaitPort( ). Con essa, il Task attende una porta. E' tuttavia probabile che il Task ne abbia più di una, e che non voglia aspettarle tutte. A questo scopo esistono i Bit di segnale. Ad ogni porta viene attribuito uno dei 32 Bit, ed ogni Task ha il proprio Bit di segnale. Un Task potrà tuttavia occupare solo un massimo di 16 Bit, perché gli altri 16 sono necessari al sistema. Al fine di poter attendere una combinazione di porte a piacere, sarà quindi necessario “mettere in alternativa” i Bit corrispondenti (addizionarne i valori) e quindi chiamare Wait() (contrariamente a

WaitPort). Un Task invia un messaggio con PutMsg(). Se tale messaggio perviene a un Task “dormiente” (uno chiamato da Wait()), esso verrà svegliato. Il Task ricevente leggerà ora il messaggio con GetMsg(). Esso terminerà la ricezione con ReplyMsg().
Per alcuni Task, come per es, Intuition, quest’ultimo è obbligatorio.

Con ciò, abbiamo appena sfiorato superficialmente l'argomento Multitasking, tuttavia dobbiamo rivolgerci ora a un’altra specialità dell'Amiga, per poter arrivare di nuovo automaticamente al Multitasking.

8.2 Screens, Windows e Gadget

Oltre alle Libraries, abbiamo altre tre questioni basilari, che è necessario conoscere, si tratta degli Screen, Window e Gadget. L’Amiga permette di inserire numerosi schermi virtuali. Su ogni schermo possono trovarsi numerose Window. La differenza di base è che uno schermo (Screen) può venire spostato sempre solo verticalmente, e che la sua dimensione preselezionata non può venire modificata. Creare uno Screen è molto facile. Si definisce una struttura, nella quale verranno memorizzati i parametri desiderati. Dopo di che si chiama OpenScreen( ). Questa funzione fornisce un puntatore all'indietro oppure zero, se qualcosa va storto. Questo puntatore deve venire quindi memorizzato nella struttura Window come parametro. I parametri essenziali di Screen sono la risoluzione (l'Amiga ne ha di quattro tipi) e la profondità, con la quale si intende il numero dei piani di Bit. Con una profondità per es' di 8, sono possibili 2 alla terza = 8 colori.

In linea di massima, una Window viene aperta come uno Screen, con la sola differenza che in questo caso è necessario definire una struttura di Window. La Window è l'elemento più potente, con il quale si avrà più spesso a che fare. Per es. non è assolutamente necessario aprire uno Screen. Se infatti nella struttura di Window si inserisce per il parametro Screen ZERO (e come tipo WBENCHSCREEN), verrà utilizzato automaticamente il Workbench-Screen. La struttura stessa è ora la seguente:

Left Edge, Top Edge:	Angolo superiore sinistro. 0,0 sarebbe a sinistra sopra
Width' Height:	Larghezza e altezza con le quali la finestra deve venire aperta
DetailPen:	Registro dei colori, con i quali devono venire disegnati i dettagli (per es. Gadget) (normalmente 0)
BlockPen:	Registro dei colori per il riempimento delle superfici,normalmente 1
IDCMPFlags:	Ved. in seguito

Flag: Ved. in seguito

FirstGadget: Puntatore al primo Gadget utente (ZERO se non presente)

CheckMark: Puntatore alla label di controllo per menu oppure ZERO se deve venire utilizzata la label di sistema

Title: Puntatore ad un testo (titolo finestra)

Screen: Il puntatore di OpenScreen, di cui la finestra deve far parte

BitMap: Puntatore ad un super Bitmap (di sollto ZERO)

MinWidth: Larghezza minima alla quale la finestra può venire ridotta

MinHeight: Altezza minima

MaxWidth: Larghezza massima

MaxHeight: Altezza massima

Type: Esistono i tipi WBENCHSCREEN e CUSTOMSCREEN. Questi ultimi vengono inseriti quando si ha necessità di un proprio Screen, diversamente si usa il Workbench-Screen.

E’ interessante notare che dopo l'apertura, questa struttura non è più necessaria.

Normalmente la si modificherà in seguito, e si aprirà un'altra finestra.

Naturalmente nessuno ha voglia di battere questa lunga struttura in ogni programma. Di conseguenza essa viene semplicemente memorizzata come file Include e ricaricata. Oltre a ciò, si ha ancora una piccola funzione, per la registrazione di quei pochi parametri che devono veramente venire modificati. Ripetiamo ancora una volta, affinché sia chiaro: tramite i dati nella struttura di apertura, Intuition genera la sua propria struttura, a cui punterà il puntatore riportato a OpenWindow(). Questo puntatore dovrà venire conservato bene, in quanto verrà utilizzato spessissimo. Passiamo ora ad analizzare ciò che è stato tralasciato:

IDCMP significa Intuition Direct Communication Message Port (IDCM) (Porta Messaggi Comunicazione Diretta di Intuition). E con ciò saremmo di nuovo in Multitasking. Tuttavia Intuition ci mette a disposizione una interfaccia utente molto comoda. Questa si occupa, fra le altre cose, dell’apertura di due porte messaggi (una per ogni direzione) quando impostiamo all'apertura il Flag IDCMP. Questi Flag sono Bit, è possibile impostare diversi Flag, mentre i Bit devono venire “messi in alternativa”.

La comunicazione fra Intuition ed una finestra si svolge ora tramite i cosiddetti Gadget. Si tratta di elementi di comando. I Gadget tipici per una finestra sono per es. il Gadget-Close (a sinistra in alto) oppure il Gadget-Size (a destra in basso) con i quali è possibile chiudere una finestra oppure modificarne la dimensione. In questo caso si tratta di Gadget di sistema (in contrapposizione ai Gadget utente).

Facciamo attenzione ad una differenza essenziale rispetto al GEM dell'Atari ST. Per quanto concerne i Gadget di sistema qui citati, Intuition esegue automaticamente le azioni che l'utente imposta con i Gadget, tranne quando il programmatore l'ha proibito. L'utente può quindi spostare una finestra oppure modificarne la dimensione, senza che il programmatore si sia preoccupato di gestire tutto ciò nel proprio programma.
Nel GEM invece accede che l'AES (che corrisponde a Intuition) segnala solo al programma che tale azione sta avendo luogo. Il programma deve quindi interrogare queste segnalazioni continuamente in un Loop, ed eventualmente chiamare le funzioni necessarie. Nel nostro caso, siamo invece molto più avanti che non in GEM, ma dobbiamo anche conoscere bene come lavorare con questo strumento molto comodo. In Fig. 81 troviamo un esempio.

```
* window1

* In questo file sono contenute diverse dichiarazioni e macro

* Osserviamo attentamente!

        incdir   ":include/"

        include  intuition/intuition.i
        include  intuition/intuition_lib.i
        include  exec/exec_lib.i
        include  graphics/graphics_lib.i


* Apertura della Intuition Library:
* --------------------------
        lea      intname,a1
        moveq    #0,d0
        CALLEXEC OpenLibrary
        tst.l    d0
        beq      abbruch
        move.l   d0,_IntuitionBase         ;Assicurare il puntatore di Base

* Apertura della Ggraphics Library
* -------------------------
        lea      grafname,a1
        moveq    #0,d0
        CALLEXEC OpenLibrary
        tst.l    d0
        beq      closeint
        move.l   d0,_GfxBase               ;Assicurare il puntatore di Base
```

```
* Apertura della Window
* --------------
        lea      windowdef,a0             ;punta alla struttura di Window
        CALLINT  OpenWindow               ;apre La Window
        tst.l    d0                       ;qualcosa non funziona?
        beq      closegraf                ;se si
        move.l   d0,windowptr             ;Assicurare il puntatore alla
                 ;Window

* Scrittura del testo nella Finestra
* ------------------------
        moveq    #100,d0                  ;Posizione X
        moveq    #50,d1                   ;Y
        move.l   windowptr,a1             ;Puntatore alla Window
        move.l   wd_RPort(a1),a1          ;Prelevamento indirizzo di Rast-
                 ;Port
        CALLGRAF Move                     ;Funzione Move to X,Y

        move.l   windowptr,a1             ;serve di nuovo Rastport
        move.l   wd_RPort(a1),a1
        lea      msg,a0                   ;Indirizzo testo
        moveq    #msglen,d0               ;sua Lunghezza
        CALLGRAF Text                     ;e output

* Attesa di un Event (qui puo' essere solo WINDOWCLOSE)
* --------------------------------------------------
        move.l   windowptr,a0             ;punta alla struttura di Window
        move.l   wd_UserPort(a0),a0       ;ora alla Message-Port
        move.b   MP_SIGBIT(a0),d1         ;Numero di Bit Segnale -> d1
        moveq.l  #1,d0                    ;in Maschera
        lsl.l    d1,d0                    ;conversione
        CALLEXEC Wait                     ;Attesa!


* Chiusura Finestra
* ------------------
        move.l   windowptr,a0             ;risveglio
        CALLINT  CloseWindow              ;chiusura finestra


* Chiusura Libraries
* --------------------
closegraf
        move.l   _GfxBase,a1
        CALLEXEC CloseLibrary

closeint
        move.l   _IntuitionBase,a1
        CALLEXEC CloseLibrary

abbruch
        move.l   #0,d0                    ;oppure fine normale
        rts

W_Gadgets        equ   WINDOWSIZING!WINDOWDRAG!WINDOWDEPTH!WINDOWCLOSE
W_Extras         equ   SMART_REFRESH!ACTIVATE
```

```
W_Title dc.b    'Titolo-Finestra',0

windowdef
        dc.w    200,50                  ;a sinistra in alto
        dc.w    300,100                 ;Larghezza, Altezza
        dc.b    -1,-1                   ;Pen dello Screen
        dc.l    CLOSEWINDOW             ;IDCMP Flag unico
        dc.l    W_Gadgets!W_Extras      ;Flag di Window
        dc.l    0                       ;nessun User-Gadgets
        dc.l    0                       ;nessun User-Checkmark
        dc.l    W_Title                 ;Titolo della Window
        dc.l    0                       ;nessun Screen proprio
        dc.l    0                       ;nessun Super Bitmap
        dc.w    100,20                  ;Dimensione Min.
        dc.w    640,200                 ;Max.
        dc.w    WBENCHSCREEN            ;Usa il Workbench Screen

intname         INTNAME                 ;Nome Intuition Lib (via Macro)
grafname        GRAFNAME                ;Nome Graphics Lib


msg             dc.b    'Hello, World! '
msglen          equ     *-msg

_IntuitionBase  ds.l    1               ;Memoria per il puntatore
_GfxBase        ds.l    1
windowptr               ds.l    1
```

Fig. 8.1: Una prima finestra di Intuition

Il programma deve portare una finestra sullo schermo e scrivere nella finestra un testo (disegnare). La finestra dovrà essere mobile sullo schermo, e la sua dimensione deve poter venire modificata. Il programma dovrà terminare quando viene clickato il box di Close della finestra.

Per fare ciò apriremo due Library, cioè Intuition e Graphics. Di solito si ha bisogno di tutte e due, dovremo inoltre assicurare immediatamente i puntatori di base, in quanto saranno spesso necessari. La Library Exec è tuttavia sempre presente. Il

```
        move.l  SysBase,a6
        jsr     _LVOxxx(a6)
```

che è obbligatorio, è nascosto nella macro

```
        CALLEXEC xxx
```

La macro CALLINT, che chiama una funzione della Library Intuition, funziona allo stesso modo. Intuition è responsabile per le questioni complesse come le finestre, al contrario Graphics è responsabile della routine di base come la tracciatura di righe, di superfici o di testi. Ambedue le istanze corrispondono all'AES, oppure VDI, del GEM dell'Atari ST.

Per poter aprire una finestra, è sufficiente una chiamata semplice, come mostrato nel listato. Tutto il resto si trova nella struttura a partire dalla Label "windowdef". Come unico Flag IDCMP abbiamo fornito CLOSEWINDOW. Ciò significa che Intuition segnala solo quando l'utente chiude la finestra. Tutti gli altri eventi (avvenimenti) vengono gestiti da Intuition da solo. Naturalmente è possibile permettere anche altri eventi, come per es. MOUSEBUTTONS oppure MOUSEMOVE, solo per citarne qualcuno.

Nella riga successiva si trova quello che Intuition deve elaborare, cioè Window_Gadget e gli extra. Lo abbiamo scritto un paio di righe sopra alle uguaglianze. Ambedue le uguaglianze avrebbero anche potuto trovarsi in "dc.l", ma il listato sarebbe divenuto troppo largo. Nella prima riga di uguaglianze si trovano i Gadget della Window, che devono venire installati. I nomi simbolici (dal file Include citato precedentemente) sono già sufficientemente chiari.

Nelle righe successive abbiamo adattato alla Window due ulteriori caratteristiche.
Smart Refresh significa che il contenuto in questione della finestra deve venire salvato, quando sta per essere ricoperto completamente o parzialmente da un altro. Non appena la finestra si ritroverà di nuovo sopra oppure la parte coperta sarà di nuovo visibile, Intuition ridisegnerà il contenuto della finestra. Contrariamente a ciò esiste anche Simple Refresh. In questo caso Intuition segnala solo i "danni". GEM, d'altra parte, è in grado di fare soltanto quest'ultima segnalazione.

Dopo l'apertura della finestra è possibile disegnare in essa. Con Move() viene impostata la posizione del disegno, con Text() viene fatta uscire una stringa a partire da tale posizione. In ambedue i casi è necessario conoscere la Rast-Port. Graphics ha sempre bisogno dell'indirizzo della Rast-Port, a cui una finestra è attribuita. Una Rast-Port è, per semplificare, una struttura che descrive le condizioni di disegno un po' più dettagliatamente di una finestra. E' possibile procurarsi il suo indirizzo con

```
move.l   wd_RPort(a1),a1
```

se in precedenza il puntatore alla finestra è stato caricato in A1. Ciò nasconde una tecnica, che vediamo nelle righe seguenti (estratto dal listato):

```
move.l    windowptr,a0    ;punta alla struttura di Window
move.l    wd_UserPort(a0) ,a0      ;ora alla Message-Port
move.b    MP_SIGBIT(a0) ,d1        ;Numero Bit Segnale -> d1
```

La nostra struttura di finestra nel listato è solo una costruzione ausiliaria. Dopo la chiamata di OpenWindow otteniamo in D0 un puntatore ad una struttura simile, che è stata preparata per noi da Intuition. Dopo ciò potremo anche buttare via la nostra struttura, modificarla, spostarla in memoria oppure utilizzarla, dopo averla modificata, per un'altra finestra.

E' importante che ci annotiamo bene il puntatore, in questo caso la variabile "windowptr". La prima delle tre righe e semplice: il puntatore viene caricato nel Registro A0. La costante "wd_UserPort" è l'Offset dall'inizio della struttura della Window ad una parola lunga all'interno della struttura. Tale parola lunga è però anch'essa un puntatore ad un'altra struttura, cioè alla porta utente Window. All'interno di questa struttura c'è un Byte, nel quale viene notato il Message-Port-Signal-Bit. E' esattamente ciò di cui avevamo bisogno.

In questo modo si accede quindi ad una delle due porte IDCM. Intuition indirizza automaticamente due di queste porte per ogni finestra. La porta di ricevimento si chiama User-Port, mentre l'invio viene fatto tramite la WindowPort. Anche se ai fini del presente esempio è forse superfluo, chiariamo comunque che se si utilizza Wait(), si dovrà dire a quali Bit di segnale (in pratica, semaforo) si vuole venire svegliati.

Dal momento che ad ogni porta viene attribuito un Bit segnale, dovremo naturalmente stabilire qual'è il nostro. Nel campo MP_SIGBIT è indicato di quale Bit si tratta (come numero di Bit) ma la funzione Wait() si aspetta una maschera in D0, nella quale tale Bit sia impostato. Di conseguenza carico il Registro D1 con il numero di Bit, quindi carico D0 con 1 e sposto quindi con LSL questo 1 al posto giusto. Non appena il Bit "suona", la finestra viene svegliata. Dal momento che noi come evento abbiamo permesso solo CLOSEWINDOW, possiamo risparmiarci i testi ulteriori e terminare il programma con la chiusura della finestra e di tutte le Library aperte.

CAPITOLO 9

Dal Task CLI alla “lcona Clickabile”

Task CLI

Task Workbench

Codice di Startup

Editor di Icone

9.1 Tipi di funzionamento dei programmi

Quasi tutti i programmi che abbiamo scritto finora potevano venire richiamati nel CLI solo con il loro nome. Questa non è la soluzione peggiore, in quanto molti comandi CLI sono anche e solo programmi di questo genere, tuttavia sappiamo che l'Amiga può fare molto di più.

Sappiamo anche che ogni CLI (con NEWCLI è possibile creare nuovi CLI addizionali) è un task. I nostri programmi erano per il task CLI, dal quale venivano chiamati, quindi erano in pratica solo dei sottoprogrammi. Mentre il nostro programma gira, cioè per es. quando aspetta un Input, anche il CLI si trova in tale sottoprogramma ed aspetta.

Il livello immediatamente successivo è un programma che può venire chiamato nel CLI con “RUN Nome”. Questo programma gira veramente come task proprio, e il CLI ridiventa libero.

Il livello più alto è formato dai programmi che possono venire lanciati dal Workbench, cioè clickando semplicemente con il Mouse sull’Icona. Lo scopo del presente capitolo è quello di creare un programma di questo genere, e naturalmente mostrare come è possibile fare ciò.

Esistono inoltre altri tipi di programmi, che possono venire lanciati sia dal CLI che dal Workbench. Penso che siamo tutti d'accordo sul fatto che ogni programma di Workbench possa girare anche sotto CLI, di conseguenza possiamo rinunciare senza problemi alla soluzione “solo Workbench”. Come vedremo in seguito, il risparmio che se ne ottiene è minimo. Riassumendo, abbiamo:

1. Sottoprogrammi CLI (chiamata con nome)
2. Task CLI (chiamata con RUN nome)
3. Task di Workbench (Click sull'lcona)
4. Combinazione di 2 e 3.

I gruppi 1 e 2 si distinguono solo per un dettaglio minimo. Il gruppo 1 utilizza per l'Input e l'Output la finestra CLI. L’Handler viene determinato con la funzione DOS Input oppure Output. Il gruppo 2 non funziona con questi Handler, ma con una sua propria finestra. I programmi del Capitolo 5 appartengono a questo gruppo. Proviamo a lanciare questi programmi con “RUN Nome”. Facciamo però attenzione ad aver sempre prima clickato con il Mouse la finestra nella quale vogliamo lavorare. E' possibile determinare immediatamente sullo schermo che dopo la chiamata, abbiamo anche “CLI2”. Ciò significa in pratica: CLI1 prepara temporaneamente per il programma (finché gira) un nuovo CLI.

9.2 Il codice di Startup

I programmi di Workbench (gruppo 3) devono adempiere ad una condizione addizionale. Essi infatti non possono venire lanciati quando si vuole, ma devono attendere, per così dire, il permesso di partenza del Workbench. Di conseguenza tali programmi, all'inizio, dovranno attendere un messaggio (il comando di Start). Quando si è terminato, si dovrà segnalare ciò al Workbench, inviando indietro al Workbench un messaggio (che ci saremo ben annotati).

Se lo stesso programma viene chiamato dal CLI, quanto descritto precedentemente non avrà naturalmente luogo. Ciò significa anche che il nostro programma deve essere in grado di distinguere da dove è stato chiamato.

ll trucco è nascosto nel cosiddetto codice Startup. Il nome non è completamente corretto, tuttavia ha preso piede in tale forma. Un codice di Start contiene sempre anche un codice di fine. Al fine di poter mettere ambedue in un solo file, che viene dapprima caricato come file Include per l'HiSoft ed il SEKA, e che viene impostato prima del programma dal Linker del Metacomco, dobbiamo utilizzare un altro trucco.

Normalmente la sequenza sarebbe:
- Codice Start
- Nostra parte di programma
- Codice fine

In pratica però, procederemo come segue:

```
        Start-Code
        jsr _main
        End-Code

_main   Nostro pezzo di programma
        rts
```

Adesso sarà chiaro anche perché in tutti i listati abbiamo messo la Label "_main" all'inizio (quando non è visibile, é contenuta nel file Include OpenDOS.i). Nel caso del Metacomco é necessario lavorare con "_main", in quanto il Linker se lo aspetta quando si linka "startup.o".

A questo punto dovrebbe essere chiaro anche che i nostri programmi devono terminare sempre con un semplice, ma importantissimo "rts". Passiamo ora alla pratica.
La Fig. 9.1 riporta il listato dello Startup.

Nel testo "ROM-Kernel-Manual, Libraries and Devices" esiste un listato piuttosto lungo in Assembler, che forma il codice di Startup per i programmi in C. Lo abbiamo abbreviato drasticamente e leggermente modificato.

Se si vogliono includere altre caratteristiche oltre a quelle in esso contenute, come per es. Alert per il caso molto verosimile che la DOS-Lib non possa venire aperta, questa è la fonte.

```
* startup.i

* Codice di Startup per programmi in Assembler. Liberamente tratto dall'
* Esempio del manuale di Kernel ROM Libraries and Devices, ma scritto
* in modo tale che tutti gli Include-File siano utilizzabili e ridotti
* al minimo indispensabile
* -------------------------------------------------------------

        incdir   ":include/"

        include "exec/exec_lib.i"
        include "libraries/dosextens.i

        movem.l  d0/a0,-(sp)              ;Salvataggio riga di comando
        clr.l    _WBenchMsg               ;per sicurezza

* Test da cosa siamo partiti
* ----------------------------------
        sub.l    a1,a1                    ;a1=0 = Task proprio
        CALLEXEC FindTask                 ;dove siamo?
        move.l   d0,a4                    ;Salvataggio indirizzo

        tst.l    pr_CLI(a4)               ;Ci troviamo sotto WB?
        beq.s    fromWorkbench            ;se si

* Siamo partiti dal CLI
* ----------------------------
        movem.l  (sp)+,d0/a0              ;Prelev. riga di comando Parms
        bra      run                      ;e avviamento

* Siamo partiti dal Workbench
* -----------------------------------
fromWorkbench
        lea      pr_MsgPort(a4),a0
        CALLEXEC WaitPort                 ;Attesa messaggio di Start
        lea      pr_MsgPort(a4),a0        ;eccolo
        CALLEXEC GetMsg                   ;preleviamolo
        move.l   d0,_WBenchMsg            ;assicurare sempre Msg!

        movem.l  (sp)+,d0/a0              ;messa a posto dello Stack.


run     bsr.s    _main                    ;chiamata del nostro programma

        move.l   d0,-(sp)                 ;salvataggio del suo Return-Code

        tst.l    _WbenchMsg               ;c'e' un WB-Message ?
        beq.s    _exit                    ;no: allora era CLI
```

```
        CALLEXEC Forbid                     ;nessuna Interruzione ora
        move.l   _WBenchMsg(pc),a1          ;prelevamento del Message
        CALLEXEC ReplyMsg                   ;e sua restituzione

_exit
        move.l   (sp)+,d0                   ;prelevamento del Return-Code
        rts                                 ;ecco fatto

_WbenchMsg       ds.l    1

                 cnop    0,2
```

Fig. 9.1: Il codice Startup

Il nocciolo della questione è costituito dalle seguenti righe:

```
        sub.l    a1,a1                      ;a1=0 = Task proprio
                 CALLEXEC FindTask          ;dove siamo?
                 move.l   d0,a4             ;Salvataggio indirizzo

                 tst.l    pr_CLI(a4)        ;Ci troviamo sotto WB?
                 beq.s    fromWorkbench     ;se si
```

La funzione di Exec “FindTask” trova l’indirizzo di una struttura di controllo del Task. Normalmente si fornisce alla funzione l'indirizzo di una stringa con il nome di task nel Registro A1. Se questo puntatore è 0, si ottiene l'indirizzo del task proprio. Insistiamo ancora una volta sul fatto che in questo caso la parola “Task” non è corretta. Tratteremo più precisamente questo argomento solo nel Capitolo 14, per il momento accontentiamoci di quanto segue: ci troviamo all’interno di un processo DOS. Un processo è qualcosa di simile ad un Task, solo di valore maggiore. “FindTask”, restituisce di conseguenza l'indirizzo della nostra BCP (Blocco di Controllo di Processo). Per chi volesse vedere questa struttura, essa è contenuta nel file Include “include/libraries/dosextens.i”.

All’interno di questa struttura troviamo una registrazione con l'Offset “pr _CLI”. Ciò significa “Puntatore all'interprete della linea di comando”. Questo puntatore è 0 se lavoriamo sotto Workbench.

Quindi, in questo caso, si tratta della Label “from Workbench” per cui avremo:

```
fromWorkbench
        lea      pr_MsgPort(a4),a0
        CALLEXEC WaitPort                   ;Attesa messaggio di Start
        lea      pr_MsgPort(a4),a0          ;eccolo
        CALLEXEC GetMsg                     ;preleviamolo
        move.l   d0,_WbenchMsg              ;assicurare sempre Msg!
```

Nella BCP con l'Offset “pr_MsgPort” abbiamo l'indirizzo che la funzione Exec “Wait-Port” vuole vedere. Questa chiamata fa attendere questo task, finché non sarà il suo turno. Immaginiamo quindi: esiste una lista di tutti i task correnti. Exec si occupa del fatto che ciascuno di essi riceva un turno l’uno dopo l'altro per un determinato periodo, in quanto è possibile far girare un solo task alla volta (abbiamo solo un 68000 nell'Amiga). Il task riceve il comando di start tramite la porta messaggi, anzi, più precisamente, solo la notizia che c’è un messaggio. Di conseguenza si deve leggere tale notizia con “GetMsg” da tale porta. Dopo ciò essa si troverà nel Registro D0. Dal momento che abbiamo ancora bisogno di questa notizia, assicuriamola nella variabile “_WbenchMsg”.

Quando il nostro Task gira sotto il Workbench, e viene “svegliato”, le seguenti righe sono interessanti:

```
run      bsr.s    _main             ;chiamata del nostro programma
         CALLEXEC Forbid            ;nessuna Interruzione ora
         move.l   _WBenchMsg(pc),a1;prelevamento del Message
         CALLEXEC ReplyMsg          ;e sua restituzione
```

Con "bsr _main" viene chiamata infine la nostra parte di programma. Dopo ciò hanno luogo gli “scontri di ritirata” cioè ciò che noi all'inizio abbiamo chiamato codice di fine. Dovremo quindi accomiatarci regolarmente dal Workbench, cosa che accede quando restituiamo il messaggio ottenuto all’inizio con lo start, tramite, “ReplyMsg”, Dal momento che anche altri task possono in teoria accedere contemporaneamente alle BCP, potrebbe anche accadere che la nostra notizia non pervenga oppure, ancora peggio, che ne derivi un caos totale.

In un sistema Multitasking, nel quale diversi task possono accedere a variabili globali comuni, c’è sempre un meccanismo che assicura ad un task per un determinato periodo il diritto di accesso esclusivo. Questa funzione, nell’Amiga, si chiama Forbid (vieta i danni). Se presa alla lettera, questa funzione è veramente pericolosa, in quanto esclude il Multitasking. Quest'ultimo resta disinserito per tutto il tempo che un task gira oppure fino a che questo non chiama Wait (attesa notizia) oppure Permit. Dal momento che noi dopo il Forbid reinviamo solo il “Reply-Message” e terminiamo, in questo caso il “Forbidding” è passabile (e comunque necessario). Dopo il “bsr_main” avevamo salvato il codice di Return del nostro programma con

```
         move.l d0,-(sp)
```

Ora dobbiamo occuparci di cosa vogliamo reinviare. Normalmente si invia zero per nessun errore. Questo “d0” dovrà naturalmente venire prelevato dallo Stack prima del RTS. Inoltre, all’inizio del programma, avevamo salvato la lunghezza e l'indirizzo di una eventuale riga di comando con

```
         movem.l d0/a0,-(sp)        ;Salvataggio riga di comando
         clr.l   _WbenchMsg         ;per sicurezza
```

ed impostato la variabile _WBenchMsg. Il riordinamento dello Stack con

```
        movem.l (sp)+,d0/a0
```

ha luogo quindi o nel CLI oppure nella diramazione di Workbench. Osserviamo ancora una volta, per finire, cosa accade effettivamente nel caso di CLI, quindi resta:

```
run     bsr.s _maln         ;Chiamata del nostro programma
        rts                 ;ecco fatto
```

Se ora assembliamo questo testo sorgente, esso dovrebbe procedere senza errori, con una sola eccezione. Tale eccezione è la mancanza della Label "_ main".

Memorizziamo ora questo file con il nome "startup.i" (in caso di necessità lo troverete anche sul dischetto allegato al presente testo), perché lo useremo al paragrafo seguente.

9.3 Demo di Multitasking

Al fine di dimostrare definitivamente che il nostro task gira veramente e fa qualcosa, scriviamo ora un programma che visualizzi continuamente in una finestra la RAM ancora libera. Il listato per fare ciò è fornito dalla Fig. 9.2.

```
        opt      l-                     ;non linkare!

* free_ram

        incdir   ":include/"

        include  startup.i              ;o come lo si e' chiamato

        include  intuition/intuition.i
        include  intuition/intuition_lib.i
        include  exec/memory.i
        include  graphics/graphics_lib.i
        include  libraries/dos_lib.i

GRAFIC  macro
        move.l   windowptr,a1       ;Indirizzo della struttura della Window
        move.l   wd_RPort(a1),a1    ;da li' alla Rast Port
        CALLGRAF \1                 ;Funzione grafica
        endm
```

```
_main

* Apertura della DOS-Library
* -------------------
        lea      dosname,a1
        moveq    #0,d0
        CALLEXEC OpenLibrary
        tst.l    d0
        beq      abbruch
        move.l   d0,_DOSBase                ;Assicurare il puntatore di base

* Apertura della Intuition Library
* --------------------------
        lea      intname,a1
        moveq    #0,d0
        CALLEXEC OpenLibrary
        tst.l    d0
        beq      closedos
        move.l   d0,_IntuitionBase          ;Assicurare il puntatore di base

* Apertura della Graphics Library
* ------------------------
        lea      grafname,a1
        moveq    #0,d0
        CALLEXEC OpenLibrary
        tst.l    d0
        beq      closeint
        move.l   d0,_GfxBase       ;Assicurare il puntatore di base

* Apertura della Window
* --------------
        lea      windowdef,a0      ;Punta alla struttura di Window
        CALLINT  OpenWindow        ;apertura della Window
        tst.l    d0                ;qualcosa non ha funzionato?
        Beq      closegraf         ;se si
        move.l   d0,windowptr      ;Assicurare il puntatore alla Window

* Loop principale
* -------------
loop    moveq    #MEMF_PUBLIC,d1            ;RAM libera
        CALLEXEC AvailMem                   ;da leggere
        move.l   d0,d2                      ;dopo  d2

        move.l   d0,d7                      ;salvataggio
        lea      buffer,a0                  ;in stringa esadecimale
        bsr      hex

        moveq    #80,d0                     ;Posizione X per testo
        moveq    #19,d1                     ;Y
        GRAFIC   Move                       ;Move TO X,Y

        lea      buffer,a0                  ;Indirizzo di testo
```

```
        addq.l   #2,a0            ;i primi 2 Nibble sono
        moveq    #6,d0            ;primo 0, gli altri 6 sono sufficienti
        GRAFIC    Text            ;Scrittura del testo


        move.l   windowptr,a0     ;Dalla nostra  Window
        move.l   wd_UserPort(a0),a0       ;al luogo di ricezione
        CALLEXEC GetMsg                   ;verificare
        tst.l    d0                       ;C'e' posta?
bne      fini                             ;Puo' essere solo CLOSEWINDOW

        move.l   #25,d1                   ;25/50 = 1/2 Secondo
        CALLDOS  Delay                    ;di attesa
        bra      loop                     ;quindi da capo

fini    move.l   d0,a1                    ;Il messaggio e' in d0
        CALLEXEC ReplyMsg                 ;rispondere

closewindow
        move.l   windowptr,a0             ;Chiusura finestra
        CALLINT  CloseWindow

closegraf
        move.l   _GfxBase,a1              ;Chiusura delle Libs:
        CALLEXEC CloseLibrary

closeint
        move.l   _IntuitionBase,a1
        CALLEXEC CloseLibrary

closedos
        move.l   _DOSBase,a1
        CALLEXEC CloseLibrary

abbruch
        moveq    #0,d0                    ;nessuna segnalazione di errore
        rts                               ;fine


* Conversione di d2.l in Stringa ASCII a partire da (a0)
* ----------------------------------------
hex     moveq    #8-1,d1                  ;tutti i Nibble
next    rol.l    #4,d2                    ;prelevamento di 1 Nibble
        move.l   d2,d3                    ;salvataggio  in d3
        and.b    #$0f,d3                  ;mascheratura
        add.b    #48,d3                   ;Trasformazione in ASCII
        cmp.b    #58,d3                   ;e'  >9 ?
        bcs      out                      ;se no
        addq.b   #7,d3                    ;diversamente deve essere A-F
out     move.b   d3,(a0)+                 ;Memorizzazione di 1 carattere
        dbra     d1,next                  ;prossimo nibble
        rts

* Conversione di d2.l -> Dec-String a partire da (a0)
```

```
W_Extras  equ   SMART_REFRESH!ACTIVATE

W_Title dc.b    ' Memoria libera attualm. ',0


windowdef
        dc.w    200,20          ;a sinistra in alto
        dc.w    220,36          ;larghezza, altezza
        dc.b    -1,-1           ;Pen delo Screen
        dc.l    CLOSEWINDOW     ;unico Flag IDCMP
        dc.l     W_Gadgets!W_Extras      ;Flag della Window
        dc.l    0               ;nessun User-Gadgets
        dc.l    0               ;nessun User-Checkmark
        dc.l    W_Title                 ;Titolo della Window
        dc.l    0               ;nessun Screen proprio
        dc.l    0               ;nessun Super Bitmap
        dc.w    100,20          ;Dimensione Min.
        dc.w    640,200         ;Max.
        dc.w    WBENCHSCREEN    ;Usa il Workbench Screen

intname         INTNAME         ;Nome delle Libs dalle Macros
grafname        GRAFNAME
dosname         DOSNAME

buffer  ds.b    8

_IntuitionBase  ds.l    1       ;Memoria per i puntatori
_GfxBase        ds.l    1
_DOSBase        ds.l    1
windowptr       ds.l    1
```

Fig. 9.2: programma che visualizza lo spazio in memoria disponibile

Spieghiamo dapprima un po' di tattica, cioè:

```
* Apertura della DOS-Library      : beq    abbruch
* Apertura della Intuition Library: beq    closedos
* Apertura della Graphics Library : beq    closeint
* Apertura della Window           : beq    closegraf

closewindow              : chiusura finestra
closegraf                : chiusura della Grafic-Lib
closeint                 : chiusura della Intuition-Lib
closedos                 : chiusura della DOS-Lib
abbruch                  : fine programma
```

Il problema è che possiamo chiudere sempre solo le Library che abbiamo precedentemente aperto. Se però apriamo più Library in sequenza, dovremo naturalmente anche sapere quali erano state aperte, al fine di poterle chiudere in caso di errore.

La soluzione è molto semplice. Le routine di chiusura vengono scritte in sequenza inversa rispetto alle routine di apertura. In caso di errore si salterà quindi alla routine di chiusura che si trova immediatamente dopo quella la cui apertura ha provocato un errore.

Dopo che sono state aperte tutte le Library di cui abbiamo bisogno, possiamo partire. Il nucleo del programma si trova nelle righe seguenti:

```
loop     moveq    #MEMF_PUBLIC,d1  ;RAM libera
         CALLEXEC AvailMem         ;da leggere
         move.l   d0,d2            ;dopo  d2
         move.l   d0,d7            ;salvataggio
         lea      buffer,a0        ;in stringa esadecimale
         bsr      hex
```

Ci procuriamo la memoria libera, attribuendo alla funzione di Exec "AvailMem" la costante "MEMF_PUBLIC". Questa costante viene definita nel file Include "include/exec/memory.i" tramite EQU. In tale file troveremo anche altri parametri utili. Proviamo ora a modificare il programma in modo tale che vengano visualizzate anche le dimensioni della Fast-RAM, della Chip-RAM ecc..

Ora ha luogo la conversione esadecimale, che conosciamo già grazie al Capitolo 5, e l'output del testo, come nell'es. del Capitolo 8. Qui, di nuovo, abbiamo solo la macro GRAFIC, che ci risparmia un po' di battitura. Anche l'attesa di un evento Intuition viene risolta in maniera diversa. Anche in questo caso, conformemente alla nostra definizione di finestra, attendiamo una sola cosa, cioè CLOSE-WINDOW.

```
move.l   windowptr,a0              ;Dalla nostra  Window
move.l   wd_UserPort(a0),a0        ;al luogo di ricezione
CALLEXEC GetMsg                    ;verificare
tst.l    d0                        ;C'e' posta?
Bne      fini                      ;Puo' essere solo CLOSEWINDOW
```

Vediamo quindi che è sufficiente leggere la porta messaggio. Se la "cassetta per le lettere" è vuota, d0 è zero. Il metodo del cosiddetto Polling non è il migliore, ma è molto efficace, in quanto, in caso di nessuna notizia, procederemo con queste righe:

```
move.l   #25,d1          ;25/50 = 1/2 Secondo
CALLDOS  Delay           ;di attesa
bra      loop            ;quindi da capo
```

Con la chiamata della routine Delay,forniamo agli altri task mezzo secondo di tempo (che per la CPU è un'eternità) per fare qualcosa da soli. Dovrebbe essere più che sufficiente visualizzare il nuovo stato della RAM ogni mezzo secondo, ma chi vuole venire informato in maniera più veloce, può naturalmente scegliere un periodo di tempo più corto.

Se si è clickato il gadget di close della finestra, otterremo un messaggio ed un salto alla Label “fini”, in essa abbiamo

```
fini     move.l   d0,a1            ;Il messaggio e' in d0
         CALLEXEC ReplyMsg         ;rispondere
```

e vale la pena di sottolinearne ancora una volta l'importanza. Dobbiamo infatti rispondere a Intuition per ogni messaggio. Ciò ha luogo molto semplicemente tramite la nota “rispedire al mittente”, cioè rinviamo il messaggio appena ricevuto.

Se il programma è stato assemblato (senza errori) ed il risultato indica “free_ram” possiamo ora andare in CLI e battere “run free_ram". Ora dovrebbe apparire la finestra con la visualizzazione. AI fine di poter eseguire un comando CLI, sarà necessario dapprima clickare in una qualunque finestra CLI. Se battiamo solo DIR, vedremo come si modifica continuamente la visualizzazione della memoria, mentre DIR gira. Il nostro programma ha ancora un piccolo errore. Noi non vogliamo immettere nulla nella sua finestra, perché quindi la finestra è attiva, e perché dobbiamo prima clickare una finestra CLI? La soluzione è motto semplice. E' sufficiente modificare la riga

```
W_Extras equ     SMART_REFRESH!ACTIVATE
```

in

```
W_Extras equ     SMART_REFRESH
```

Con ciò la finestra non è più attiva, e il suo titolo verrà scritto in grigio. Naturalmente, è comunque possibile clickarla, se si preferisce la forma attiva.

9.4 Icone ed Editor per Icone

Al fine di poter lanciare un programma da Workbench, dotato di un codice di Startup normale, è necessaria ancora una Icona. Affinché una Icona sia visibile, essa dovrà:
a) essere disponibile (logicamente) e
b) trovarsi in una directory che ha essa stessa una Icona, cioè è visibile come cassetto.
Il sistema più semplice per costituire un cassetto di questo genere, è quello di duplicare da Workbench il cassetto “Empty”. E’ comunque anche possibile battere semplicemente in CLI:

```
copy empty.info test.info
```

Torniamo ora per un attimo al Workbench, dove non vedremo la nuova Icona. Chiudiamo quindi la finestra disco e riapriamola. Ora vedremo il cassetto Empty un po’ spostato ed il cassetto Test diventerà visibile

Ora abbiamo bisogno di una Icona Programma. Prendiamone una disponibile, anche se non tutte sono adeguate. L'Amiga conosce diversi tipi di Icone. Quali esse siano e quale significato esse abbiano, lo impareremo automaticamente quando lanceremo l'Editor di lcone. Per noi è importante sapere che i programmi devono essere del tipo TOOL.

Lo stesso IconED è per es. adeguato. Ipotizziamo di avere già costituito la directory (il cassetto) “test” e che il nostro programma si chiami “free_ram“. Quindi copiamo dappri-ma il programma con

```
copy free_ram :test/free_ram
```

Dopo ciò copiamo una Icona (IconED si trova nel “raccoglitore” del sistema)

```
copy :system/iconed.info :test/free_ram.info
```

Ora, sul Workbench, dovremmo trovare una lcona “test” nel cassetto, simile a quella di lconED, ma riportante il titolo “free_ram”. Possiamo clickarla fiduciosamente, “free_ram” partirà.

Se vogliamo ora fornire alla nostra Icona un aspetto adeguato, chiamiamo IconED.
Nel menu disco scegliamo LOAD e forniamo come risposta alla richiesta di testo

```
:test/free_ram
```

cioè sempre l'intero nome di percorso. La modifica è semplice ed auto documentante.
E’ sufficiente trovare i diversi punti del menu, ciò che è importante sapere è che per di-segnare/modificare un'Icona, è sempre necessario selezionare il menu Color e da esso il colore adeguato. La cancellazione è possibile con il colore di sfondo. Sia il disegno che la cancellazione vengono effettuati con il Mouse. Il tasto di sinistra preme la matita sulla “carta”.

E' possibile anche rinunciare al procedimento di copiatura e costruire da soli un’Icona nell'Editor. Dovremmo quindi fornire il nome corretto alla richiesta di “Save”, per cui nel nostro esempio forniremo di nuovo

```
:test/free_ram
```

Per il resto, non ci sono altri problemi. Per essere sinceri, potranno nascere le lcone più inverosimili, ma è sempre solo il file Info che viene editato. Al programma non accade nulla.

9.5 Trasformazione di parole lunghe in stringhe decimali

La soluzione per la trasformazione da esadecimale a decimale è offerta in Fig. 9.3.
Per la routine bindec avremo bisogno di una divisione in parole lunghe. Tale divisione è disponibile in quasi tutti i testi che si occupano del 68000, quindi, tanto per cambiare, osserviamo qualcosa di diverso.

Il procedimento è vecchio come il mondo, io personalmente l'ho già incontrato nell'interprete BASIC della prima ora. Su di un 68000, esso gira in modo particolarmente veloce, in quanto quest'ultimo lo supporta con il suo tipo di indirizzamento raffinato. Il principio è la divisione tramite sottrazione continua. Ipotizziamo che il numero sia 321. Da esso io potrò sottrarre 3 volte 100: al quarto tentativo otterrò un numero negativo.
Ora conto quante sottrazioni faccio per arrivare ad un numero negativo (4) sottraggo di nuovo 1 e ottengo la prima cifra (il 3). Ora dovrò di nuovo aggiungere 100 al numero rimanente, per cui resta 21. Da questo 21 sottraggo ora gruppi di 10. Al terzo tentativo, ciò porterà ad un numero negativo, per cui -1 fornisce la cifra 2.

La nostra parola lunga, tuttavia, va bene per numeri di circa +/-2 miliardi, per cui la gamma dei numeri non dovrebbe partire da 100 ma da 10 milioni. Questa colonna è illustrata nella tabella. Attenzione: l'accesso alla tabella è sempre più veloce della determinazione della cifra con il calcolo. La cosa nuova in questo caso, è che faccio attenzione al segno. In caso di numeri negativi ne consegue per forza uno positivo (neg.l) e questo fatto viene annotato in d3.

Più tardi, dopo che gli zeri iniziali sono stati sostituiti da spazi bianchi, nel Buffer verrà scritto anche un segno meno. Fare comunque attenzione al fatto che qui è necessario fare uscire dieci posizioni tramite _LVOWrite

```
* Conversione di d2.l -> Dec-String a partire da (a0)
decl            clr.b    d3                 ;0 = numero positivo
                tst.l    d2                 ;Numero positivo
                bpl      plus               ;se si
                neg.l    d2                 ;altrimenti Trasformazione
                move.b   #1,d3              ;marcatura numero negativo
plus            moveq    #7,d0              ;Converti 8 cifre
                lea      buffer+1,a0        ;+1 per spazio per il segno
                lea      pwrof10,a1         ;Tabella
next            moveq    #'0',d1            ;Inizia con la cifra '0'
decl            addq     #1,d1              ;Cifra + 1
                sub.l    (a1),d2            ;c'e' ancora qualcosa?
                bcc.s    dec                ;se si
                subq     #1,d1              ;cifra corretta
                add.l    (a1),d2            ;poiche' anche
                move.b   d1,(a0)+           ;Numero -> Buffer
                lea      4(a1),a1           ;prossima potenza di 10
```

```
                dbra     d0,nex            ;per 8 cifre
                lea      buffer,a0         ;Soppressione dello Zero e del
segno
rep             move.b   #' ',(a0)+        ;zeri iniziali
                cmp.b    #'0',(a0)         ;sostituire
                beq      rep               ;con spazi bianchi
                tst.b    d3                ;il numero era negativo?
                Beq      done              ;se no
                move.b   #'-',-1(a0)       ;Altrimenti spostamento
done     rts

pwrof10  dc.l     10000000
         dc.l     1000000
         dc.l     100000
         dc.l     10000
         dc.l     1000
         dc.l     100
         dc.l     10
         dc.l     1
```

Fig. 9.3: Routine per la rappresentazione decimale di “parole lunghe”

CAPITOLO 10

Vista d'insieme dei comandi del 68000

Gli interni

Il background

In questo capitolo presentiamo i comandi del 68000 in una visione d'insieme. Nell'Appendice A1 troveremo per ogni singolo comando la forma sintattica e i tipi di Indirizzamento permessi, in essa viene anche descritto quale lunghezza di operandi (Byte, Parola, Parola lunga) è permessa caso per caso. In questo capitolo ci occupiamo primariamente della tematica “cosa è disponibile e a cosa serve”.

10.1 Comandi di Trasferimento

Comando	Significato
EXG	Scambio di contenuto di registri
LEA	Caricamento di un registro
LINK	Aumento dello Stack locale
MOVE	Trasferimento (copiatura) di dati
MOVEA	Trasferimento (copiatura) di indirizzi
MOVEM	Trasferimento (copiatura) d1 più registri
MOVEP	Trasferimento (copiatura) di dati alle periferiche
MOVEQ	Trasferimento (copiatura) di costanti “Quick”
PEA	Portare un indirizzo allo Stack
SWAP	Scambio di parole di un registro
UNLK	Diminuzione dello Stack locale (ved. LINK)

Ciò che sicuramente salta all’occhio sono le molte varianti del comando MOVE. Con esse è possibile verificare una volta per tutte l'assemblatore. I buoni assemblatori accettano anche un MOVE, laddove si sarebbe dovuto scrivere MOVEA. Gli assemblatori molto buoni forniscono anche le segnalazioni di quando non si è programmato in modo ottimale, cioè quando per es. si è scritto MOVE invece di MOVEQ.

10.1.1 LINK e UNLINK

Particolarmente degni di citazione sono senza dubbio i comandi LINK e UNLK (Unlink). Con questi comandi il 68000 viene preparato in modo particolare ai compiti dei compilatori dei linguaggi evoluti. Anche qui c'è sempre il problema che, in procedure e funzioni, devono venire create delle variabili locali che esisteranno solo per tutto il tempo in cui la procedura (funzione) è attiva. Di conseguenza è comune porre tali variabili nello Stack.

L'ideale è quando è possibile riservare il campo di Stack adeguato con un solo comando, e quindi liberarlo di nuovo in modo altrettanto semplice. Ciò viene offerto dai comandi LINK e UNLK. Quando una procedura (sottoprogramma) chiama un'altra procedura e questa ne chiama una terza ecc., e ciascuna di queste procedure funziona con il proprio Stack locale, ne deriva per cosi dire una lista concatenata, in inglese "linked list". Da ciò derivano anche i nomi LINK e UNLK. Osserviamo esattamente come funziona. La sintassi del comando è

```
LINK An,#Distanza indirizzo
```

Un esempio:

```
LINK    A6,#30
```

in questo caso A6 viene dapprima messo nello Stack, cioè viene eseguito il comando

```
move.l  a6,-(sp) (Passo 1)
```

Ora il puntatore allo Stack viene copiato nel registro appena salvato, cioè

```
move.l  sp,a6 (Passo 2)
```

Quindi la distanza di indirizzamento viene aggiunta al puntatore allo Stack, e ciò significa

```
 add.l  #Adr_Dist,sp
```

Con ciò avremmo lo Stack locale per un sottoprogramma. Normalmente si sceglie la distanza di indirizzamento, negativa, dal momento che come è noto lo Stack cresce verso indirizzi decrescenti. Con UNLK, viene ricostituito lo stato iniziale. Praticamente UNLK funziona come

```
move.l  a6,a7
move.l  (sp)+,a6
```

Per il programma principale oppure per il sottoprogramma che stava chiamando, il registro di indirizzamento ed il puntatore allo Stack hanno di nuovo il loro valore originario.

Oltre a ciò, c'è da fare attenzione al fatto che il registro di indirizzamento contiene, dopo il comando LINK, una copia del puntatore allo Stack. Con ciò un sottoprogramma potrà accedere molto facilmente a dati del programma chiamante, se quest'ultimo ha in precedenza messo nello Stack tali dati.

10.2 Comandi aritmetici

Comando	Significato
ADD	Addizione di dati
ADDA	Addizione di indirizzi
ADDI	Addizione di una costante
ADDQ	Addizione di una costante “Quick”
ADDX	Addizione con Bit di riporto
CLR	Cancellazione di un operando
CMP	Confronto di due dati
CMPA	Confronto di due indirizzi
CMPI	Confronto con una costante
CMPM	Confronto di due dati in memoria
DIVS	Divisione con segno
DIVU	Divisione senza segno
EXT	Ampliamento che tiene conto del segno
MULS	Moltiplicazione con segno
MULU	Moltiplicazione senza segno
NEG	Negazione
NEGX	Negazione con Bit X
SUB	Sottrazione di dati
SUBA	Sottrazione di indirizzi
SUBI	Sottrazione di una costante
SUBQ	Sottrazione di una costante “Quick”
SUBX	Sottrazione con X-Bit (prestito)
TST	Testa gli operandi rispetto a zero
ABCD	Addizione di cifre BCD
NBCD	Negazione di cifre BCD
SBCD	Sottrazione di cifre BCD

Anche in questo caso, i buoni assemblatori possono brillare per le loro caratteristiche. Per lo meno, dovrebbero essere daccordo con ADD, anche quando sarebbe stato necessario immettere ADDA oppure ADDI. Lo stesso vale per CMP e SUB.

Il 68000, in questo caso, è particolarmente utile grazie a due caratteristiche. La prima è che le operazioni aritmetiche sono possibili su una larghezza di 32 Bit.

Da questo punto di vista, anche il 68000 è un vero “32 Bit”. Il suo Bus di dati è comunque ampio solo 16 Bit, cosa che conduce al fatto che le parole lunghe vengono trasportate in “due porzioni”, ma ciò ci tocca solo in maniera secondaria. Primariamente, è importante il fatto che con ciò le operazioni matematiche sono sostanzialmente più semplici da programmare che non su CPU che permettono una larghezza di operandi di 16 o addirittura solo di 8 Bit. Secondariamente si dovrà naturalmente anche fare attenzione al fatto che i dati vengono tenuti nei registri il più a lungo possibile, per cui non ha più luogo nemmeno il trasferimento, relativamente lungo nel tempo,
tramite il Bus dei dati. Ciò tuttavia non dovrà venire sopravvalutato, se si tiene conto della elevata velocità del 68000. E' solo in caso di routine a calcolo intenso che ci si potrà accorgere di ciò.

10.2.1 Aritmetica BCD (Binary-Coded-decimale)

L'aritmetica BCD del 68000 sarà particolarmente amata da chi ha dovuto programmare qualcosa di simile su di un'altra CPU. Una cifra BCD si trova sempre in un mezzo Byte (4 Bit). Dal momento che è noto che con ciò è possibile rappresentare i numeri da 0 a 15, e che tuttavia sono validi solo da 0 a 9, avremo alcuni problemi con dei numeri maggiori. Per le altre CPU si dovrà controllare questo caso tramite il cosiddetto Half-Carry-Flag; nel nostro caso, invece, è sufficiente sommare. Dal momento che come dimensione di operandi sono permessi sempre e solo dei Byte, avremo un superamento dopo 99. Questo tuttavia andrà automaticamente nel Flag X e verrà, altrettanto automaticamente, aggiunto. Ecco un esempio per l'addizione di due cifre a sei posizioni in tre Byte cadauna.

Numero	Valore	In memoria agli indirizzi
1	123456	12 a 1001, 34 a 1002, 46 a 1003
2	654321	65 a 2001, 43 a 2002, 21 a 2003

Come al solito si dovrà cominciare da destra (con le unità) per fare l'addizione. Di conseguenza, come tipo di indirizzamento della memoria, è permesso qui anche “ARI con predecremento”. Il programma risulterebbe quindi come segue:

```
move    #1004,a1
move    #2004,a2
move    #4,CCR

ABCD    -(a1),-(a2)
ABCD    -(a1),-(a2)
ABCD    -(a1),-(a2)
```

Si dovrà tuttavia fare attenzione a quanto segue:
1. A causa del predecremento, il Byte con le unità dovrà trovarsi ad un indirizzo dispari

2. Al fine di evitare l'addizione, la prima volta, di un Flag X casuale, lo si dovrà cancellare.
3. Al fine di poter riconoscere un risultato zero, sarà necessario impostare preceden temente il Flag Z.

I punti 2 e 3 possono venire risolti molto semplicemente con l'istruzione “move #4,CCR”. Ciò che non viene risparmiato è la necessità di garantire in precedenza che le cifre siano cifre BCD (richiesta al caricamento). I valori maggiori di 9 verranno infatti sommati in maniera errata.

10.3 Comandi logici

Comando	Significato
AND	AND logico
ANDI	AND logico con una costante
EOR	XOR logico
EORI	XOR logico con una costante
NOT	NO logico (complemento di 1)
OR	OR logico
ORI	OR logico con una costante

L'unica annotazione a questi comandi è che, come noto, essi hanno effetto Bit per Bit.

10.4 Comandi Bit

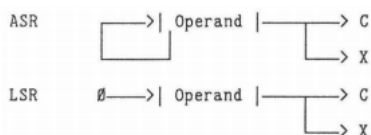
Comando	Significato
BCHG	Cambia (inverti) un Bit
BCLR	Cancella un Bit
BSET	Imposta un Bit
BTST	Verifica un Bit
TAS	Verifica e imposta il Bit 7 di un operando di Byte

I comandi Bit mantengono sempre lo stato precedente nel Flag Z, quindi eseguono due operazioni. Questa caratteristica, ed in particolare quella di TAS, sono un buon esempio per le capacità particolari del 68000, cioè per il supporto del Multitasking. Con il

comando TAS viene interrogato il Bit 7 di un operando di Byte in memoria ed il risultato viene annotato come al solito nel Flag Z. Dopo ciò viene scritto un 1 nel Bit 7. L'intera sequenza, cioè la lettura dell'operando, l'interrogazione e la riscrittura, è inseparabile, cioè non può venire interrotta tramite un Interrupt. Tali comandi non separabili sono molto importanti in un sistema Multitasking. Si potrebbe citare per es. compiti quali il cambio di processo, la sincronizzazione di processi ecc..

10.5 Comandi di rotazione e scorrimento

Comando	Significato
ASL	Spostamento aritmetico verso sinistra
ASR	Spostamento aritmetico verso destra
LSL	Spostamento logico verso sinistra
LSR	Spostamento logico verso destra
ROL	Rotazione a sinistra
ROR	Rotazione a destra
ROXL	Rotaziohe con X-Bit a sinistra
ROXR	Rotazione con X-Bit a destra

A questo proposito c'è da notare che con un comando è possibile shiftare/ruotare fino ad un massimo di 31 Bit. Le altre CPU, con una istruzione, eseguono la stessa cosa solo per un Bit. Fra ASL e LSL non c'è in pratica nessuna differenza, mentre c'è fra ASR e LSR, come mostrato nella figura seguente:

Il disegno precedente dovrà mostrare che,con ASR, il Bit del segno viene sempre ripristinato, cioè non viene shiftato (spostato). In caso di LSR, al contrario, viene aggiunto un Bit zero. In caso di spostamento verso sinistra, sia per ASL che per LSL, viene fornito un Bit di zero da destra.

10.6 Comandi di gestione del programma

Comando	Significato
Bcc	Diramazione condizionale
BRA	Diramazione sempre
BSR	Diramazione ad un sottoprogramma
CHK	Controllo registro dati rispetto al limite 0 e a un altro
DBcc	Loop condizionale
JMP	Salto ad un indirizzo
JSR	Salto ad un sottoprogramma
NOP	Nessuna operazione
RESET	Reset della periferica
RTE	Ritorno da una eccezione
RTR	Ritorno con caricamento del flag
RTS	Ritorno da un sottoprogramma
Scc	Impostazione di un Byte condizionale
STOP	Arresta il programma a
TRAP	Va in eccezione
TRAPV	Va in eccezione quando il flag V è impostato

A questo punto sarebbe da notare la differenza fra i comandi di diramazione e quelli di salto. I primi sono sempre relativi al PC attuale, tuttavia sono limitati ad una gamma di indirizzi di +/—32 Kbyte. I comandi di salto JMP e JSR spaziano per tutto il campo di indirizzi di 16 Mbyte.

CHK verifica un registro dati rispetto a due limiti, cioè 0 e un limite indicato nell'operando. Se il test risponde, viene emessa una eccezione. Con ciò è molto semplice realizzare il controllo di una gamma, per es. degli indici delle matrici.

Ora, se abbiamo contato bene, si trattava di 56 comandi. Rispetto ad altre CPU, ciò è relativamente poco, ma non facciamoci ingannare. La maggior parte dei 12 tipi di indirizzamento può venire utilizzata sugli operandi di sorgente e di destinazione, da cui derivano più di 1000 varianti. Le altre CPU danno ad alcune di queste varianti dei nomi propri di comando, ma con ciò non si aumentano le prestazioni, bensì solo il numero di dati che il programmatore deve memorizzare.

10.7 Conoscenze di base

Nella presente sezione dobbiamo occuparci un attimo delle conoscenze di base relative al 68000. Esse in realtà non sono assolutamente necessarie per la programmazione, ma è sicuramente molto meglio sapere perché si fanno determinate cose.

Durante la lettura del presente testo avrete probabilmente già notato che io sono un estimatore del 68000, e la mia stima è basata su una esperienza di lunghi anni nella programmazione di altre CPU.

Il 68000 e il primo microprocessore il cui gruppo di comandi è basato su quello dei minicomputer. Se si pensa che computer così moderni come l'Amiga con la sua memoria principale di 1-4 Mbyte (e 256 Kbyte nella ROM) è superiore ai mini degli anni 70, che un sistema 68020 nel frattempo è già in grado di battere una VAX, in certi settori, è chiaro che questo gruppo di comandi è l'unica alternativa. Gli elaboratori di elevate prestazioni hanno bisogno infatti di Software di sistema molto complessi, che sarà possibile produrre con sicurezza ed efficacia solo quando la CPU offre delle basi solide.

10.7.1 La struttura interna del 68000

In linea di massima una CPU è sempre composta da una unità di calcolo, si dice anche “Control Logic” e “Arithmetic Logic Unit” (ALU). L'unita di controllo è composta dai

- Registri di comando e
- Decodificatori dei comandi

Il decodificatore dei comandi fornisce i suoi risultati alla unità esecutiva, nella quale incontra per es. delle funzioni ALU oppure una serie di registri. Il nocciolo della questione è ora la codifica dei comandi. I primi microprocessori, come per es. il leggendario F8, erano in linea di massima nient'altro che componenti logici programmabili.
Determinate sequenze di Bit in input producevano altre sequenze di Bit in output. In questo senso anche i comandi non erano altro che sequenze di Bit, che venivano decodificate tramite l’Hardware. E' forse ancora possibile utilizzare oggi la stessa tecnica con una CPU ad 8 Bit, ma in presenza di un gruppo di comandi così potente come quello del 68000, essa ci condurrà immediatamente in un vicolo cieco. Infatti non è più possibile orientarsi nell'Hardware, che ha bisogno di molto spazio e non è quasi modificabile. L’unica via di uscita e il cosiddetto “Micro-code”.

Semplificando: i comandi del 68000, come li conosciamo noi, sono già un linguaggio evoluto dal punto di vista della CPU. Tramite un programma, la CPU traduce questi comandi macro provenienti dall’esterno in una sequenza di comandi micro interni.
L'unita di controllo della CPU, di conseguenza, non è predisposta come una logica Hardware, bensì come programma. Tale programma si trova in una zona della ROM sul Chip CPU.

Dal momento però che ci volevamo rivolgere all'Hardware della CPU, nasce immediatamente un problema. Per ogni comando infatti devono venire impostati moltissimi Bit, per es. 64 solo se ci si vuole rivolgere a due registri. Se si crea il micro-code molto stretto (per es. con una larghezza di 4 Bit) si avrà bisogno di moltissimi microcicli, quindi di molto tempo per l'esecuzione di un comando macro. In questo caso si parla di microcode verticale. Se si rende il micro-code più largo orizzontalmente, si avranno meno cicli. In questo caso tuttavia il tempo del decodificatore aumenterà. Ora, è possibile trovare un compromesso fra i due tipi (verticale o orizzontale) oppure, meglio, combinare insieme l'uno con l'altro.

Questo é esattamente ciò che viene fatto dal 68000. Esiste un micro-code-ROM (larghezza comando 9 Bit) ed un Nano-code-ROM (70 Bit). I micro-code sono in linea di massima solo dei puntatori ai nano-code, da cui deriva una decodifica veloce, simile a quando, invece di sfogliare tutte le pagine di un libro, si esamina dapprima attentamente l'indice.

Ciò nonostante, la pura logica Hardware è più veloce, dal momento che in essa non è presente nessun tempo di ricerca. Al fine di ovviare a questo svantaggio, si è escogitato un sistema,chiamato Prefetch. Prefetch significa "prendere prima". Un comando può essere composto da un massimo di 5 parole. Il ciclo totale è

- Prendere
- Decodificare
- Eseguire

Se si sovrappongono questi passaggi, si risparmierà naturalmente del tempo. Praticamente il 68000, durante l'elaborazione di un comando, prende già la parola di comando successiva e quella successiva ancora. Quindi viene sempre letta una nuova parola mentre un'altra è in corso di elaborazione.

10.7.2 Modi utente e supervisore

Come già accennato, nel 68000 esistono i modi utente e supervisore. Il vantaggio di questa separazione è chiaro. Le routine di base del sistema operativo non possono venire disturbate da un errore nel programma utente. Ciò inoltre rende possibile a noi utenti riuscire a rintracciare tali errori. Infatti, come potrà per es. un Debugger mostrarci i contenuti dei registri, se le routine di sistema operativo necessarie sono state distrutte dal programma contenente errori?

La differenza tra questi due modi viene determinata da un solo Bit, cioè dal Bit S del registro di stato. Se l'Amiga viene avviato con un "avviamento a freddo" (accensione) oppure "avviamento a caldo" (tasto di reset) esso si troverà automaticamente in modo supervisore. Il passaggio dal modo supervisore al modo utente può avere
luogo tramite:
- RTE
- Modifica del Bit supervisore (MOVE # K,SR / ANDI # K,SR etc.)

Si passa invece dal modo utente al modo supervisore tramite
- Interrupt
- Comando di Trap
- Eccezione (per es. errore di indirizzamento).

Nel modo utente sono disponibili tutti e 8 i registri di dati, i 7 registri di indirizzi da A0 ad A6, il puntatore allo Stack (USP) ed il PC. Non avrà mai luogo un accesso al puntatore allo Stack del supervisore (SSP).

L'accesso al registro di stato è limitato. Infatti il registro di stato è composto dal Byte supervisore, chiamato anche Byte di sistema, e dal Byte utente (CCR = Condition Code Register). E' possibile un accesso completo al CCR, mentre per il Byte di sistema è possibile accedere solo in lettura, se si è in modo utente.

Trace-Bit

Il trace-bit ci offre un aiuto particolare. Se questo Bit (Bit 15 nel registro di stato) è impostato, il 68000 dopo ogni comando va in una eccezione, cioè salta all'indirizzo che si trova nel vettore trace (vettore 9 all'indirizzo $24). Con ciò è possibile l'elaborazione passo passo. La routine alla quale punta il vettore trace potrà per es. visualizzare i contenuti di registro. In altre parole: la parte più difficile di un Debugger è già incorporata nel 68000. E' facile immaginare che, senza questa caratteristica, sarebbe molto oneroso programmare il Tracing. Infatti si dovrebbe sempre mettere nel codice da verificare un salto alla routine trace del comando successivo, quindi sostituire tale comando con quello originale, spostare il salto di trace etc. Per fare ciò, inoltre, è necessario sapere quanti Byte occupa ciascun comando, quindi bisognerebbe far girare contemporaneamente un disassemblatore.

10.7.3 Le Eccezioni

Abbiamo già applicato molto spesso le eccezioni senza accorgercene (Exec lavora con noi) per cui è giunto il momento di approfondire meglio questa importante caratteristica.

Con la parola inglese Exception si intende eccezione, e il 68000 può andare in eccezione in tre modi:
1. A causa di segnali esterni (Interrupt, errore del Bus, Reset)
2. A causa di errori (per es. errori di indirizzo)
3. "Con intenzione"

Il terzo caso viene usato spesso da Exec, che emette le eccezioni con i cosiddetti comandi Trap. Se anche noi vogliamo lavorare in questa maniera, dobbiamo procurarci un vettore exception da Exec. Diversamente ci scontreremo con il sistema operativo.

Chi proviene dall'Atari ST oppure dal Macintosh, e sente la mancanza di un uso frequente di Trap, tenga presente che in un sistema multitasking il sistema operativo è piuttosto complicato. Il reset o l'interrupt sono molto più facili da spiegare, in quanto è sufficiente comandare i loro relativi input del 68000.

Errori di Bus

L'errore di Bus è già qualcosa di più complicato. Nel 68000 esistono dei collegamenti Hardware (MMU) che si occupano principalmente del fatto che gli indirizzi generati dal 68000 si rivolgano ai Chip di memoria giusti. Di conseguenza un MMU ha sempre un'uscita-errore, (Fault), che viene attivata quando ci si rivolge ad un indirizzo non esistente.

Cosa accade con una eccezione?

Non è facile rispondere a questa domanda. In effetti esistono due tipi di eccezioni.
In linea di massima il contatore di programma (PC) ed il registro di stato (SR) vengono sempre posti nello stack. In caso di un errore di Bus oppure di indirizzo, servono ancora tre informazioni, cioè
- Il codice del comando in corso di elaborazione
- L'indirizzo al quale si stava accedendo
- La super-parola dello stato

Nella super-parola dello stato, sono rilevanti i Bit da 0 a 4, cioè

Bit 0 ... 2:	Codice di funzione
Bit 3:	0 = gruppo 2, 1 = gruppo 1
Bit 4:	0 = il ciclo di scrittura è stato interrotto
	1 = ciclo di lettura

A questo punto c'è ancora da chiarire che i codici di funzione sono tre output del 68000 con i quali la CPU indica alla MMU che cosa sta facendo. In essi ha luogo essenzialmente la differenziazione fra gli accessi a dati applicativi, programmi applicativi, dati supervisore e programmi supervisore. Ciò significa che la CPU naturalmente sa se si trova in modo supervisore oppure in modo utente, e se il PC sta puntando ad una parola comando oppure ad un dato.

I gruppi 0, 1 e 2 hanno il seguente significato. Pensiamo un attimo a cosa accadrebbe se il 68000 si trovasse in elaborazione di una eccezione e gli si presentasse un'altra eccezione.

Per questi casi esistono le priorità. La priorità maggiore è detenuta dal gruppo 0, quindi seguono l'1 e il 2. I raggruppamenti vengono effettuati come segue:

Gruppo 0	Reset Errore Bus Errore indirizzo
Gruppo 1	Trace Interrupt Comando non ammesso Trap $Axxx,Trap $Fxxx Infrazione del privilegio
Gruppo 2	Trap #n Trapv CHK Divisione per 0

Se osserviamo questo elenco come elenco unico, vediamo immediatamente che reset ha la priorità maggiore, mentre la divisione per 0 ha la priorità più bassa. Se per es. durante una eccezione ne arriva un'altra con una priorità maggiore, il programma in corso viene interrotto, viene elaborata la routine con priorità maggiore, quindi viene ripresa la routine interrotta.

Eccezioni in Amiga

Come già detto, dopo una eccezione,alcune informazioni vengono salvate nello Stack, quindi il PC viene caricato con l'indirizzo che si trova nel vettore corrispondente. Ciò ha effetto come un salto a tale indirizzo. Di conseguenza in ogni vettore dovrebbe già esserci memorizzato qualcosa, in modo che il 68000 non "giri a vuoto". Nel vettore 0 è contenuto il valore che il puntatore allo stack dovrà assumere dopo il reset, nel vettore 1 quello del primo stato del PC. Questi vettori sono obbligatori e devono essere disponibili al momento dell'accensione o del reset. Poiché al momento dell'accensione una RAM è naturalmente vuota, sarà l'Hardware a dover fare in modo che in tali indirizzi ci sia "un po' di ROM". I vettori rimanenti possono venire occupati a piacere dal programmatore del sistema. In ogni caso, non tutti saranno necessari. I vettori rimasti liberi dovrebbero tuttavia venire caricati con uno (stesso) indirizzo. La routine su tale indirizzo può cominciare e terminare semplicemente con RTE oppure contenere una piccola segnalazione, tipo "vettore nr. x libero". Invece di questo messaggio l'Amiga ha la sua amata Guru-Meditation.

CAPITOLO 11

Struttura dei dati dell'Amiga

Trucchi con le macro
e
strutture dei dati

Chiave per la programmazione in Amiga

11.1 Strutture dei dati, chiave per la programmazione in Amiga

La prima struttura di dati è stata affrontata nel Capitolo 8, Fig. 8.1. In Fig. 11.1 ritroviamo di nuovo la struttura Window:

```
windowdef
        dc.w    200,50                  ;a sinistra in alto
        dc.w    300,100                 ;Larghezza, Altezza
        dc.b    -1,-1                   ;Pen dello Screen
        dc.l    CLOSEWINDOW             ;IDCMP Flag unico
        dc.l    W_Gadgets!W_Extras      ;Flag di Window
        dc.l    0                       ;nessun User-Gadgets
        dc.l    0                       ;nessun User-Checkmark
        dc.l    W_Title                 ;Titolo della Window
        dc.l    0                       ;nessun Screen proprio
        dc.l    0                       ;nessun Super Bitmap
        dc.w    100,20                  ;Dimensione Min.
        dc.w    640,200                 ;Max.
        dc.w    WBENCHSCREEN            ;Usa il Workbench Screen
```

Fig. 11.1: Struttura di NewWindow

Se rinunciamo per una volta ai passi obbligatori all'inizio di un programma (aperture di Libraries) e a quelli della sua chiusura (chiusura delle Libraries) otterremo che un programma tipico per Amiga non è altro che una continua ripetizione di due passi,cioè
- Definizione della Struttura dei dati
- Chiamata della funzione con la struttura come parametro

Nell'esempio del Capitolo 8, volevamo aprire una Window di Intuition. L'apertura vera e propria è stata risolta con una sola riga, mentre il lavoro è contenuto nelle righe di cui a Fig. 11.1. Scrivendo “Hello World!” abbiamo avuto un compilo relativamente semplice. Avremmo infatti potuto prendere un altro Font (tipo di carattere) quindi avremmo dovuto scrivere:

```
MyFont  dc.l    font_name
        dc.w    TOPAZ_SIXTY
        dc.b    FS_NORMAL
        dc.b    FPF_ROMFONT
```

Anche questa sarebbe una struttura. Avremmo anche potuto definire il nostro Screen, e anche ciò necessita di una struttura, avremmo potuto voler lavorare con dei menu di Pull-down (molte strutture) etc.

Quindi come sempre, qualunque cosa si voglia programmare, non è possibile prescindere dalle strutture. Purtroppo queste strutture sono talvolta molto complesse, in ogni caso sono numerose ed un piccolo errore (dc invece di dc.l) conduce per lo meno a dei risultati molto strani, e per lo più ad un fallimento.

11.2 File include

Le strutture più importanti sono contenute nell'Appendice, non è possibile elencarle tutte nel corso del presente testo. Tutte le strutture sono tuttavia contenute nei file Include, purtroppo in essi sono rappresentate in una maniera poco comprensibile e, ancor peggio, in una forma molto poco pratica per i programmi in Assembler (nonché per la maggior parte dei programmi in C).

Il fatto è che le strutture sono state scritte originariamente in C, quindi tradotte in Assembler. I programmatori pigri avranno naturalmente utilizzato delle macro, producendo il risultato rappresentato in Fig. 9.2.

```
STRUCTURE       macro                       ;Sistema per alcuni Assembler
\1              set     0                   ;per dare un nome
Offset          set     \2                  ;e fare in modo che
                                            ;cominci da 0
                endm

BYTE            macro
\1              equ     Offset
Offset          set     Offset+1            ;L' LC conta in Byte
                endm

WORD            macro
\1              equ     Offset
Offset          set     Offset+2            ;Una parola ha 2 Byte
                endm


LONG            macro
\1              equ     Offset
```

```
Offset          set     Offset+4        ;Una parola lunga ha 4 Byte
                endm

ULONG           macro                   ;Anche Unsigned è lungo
\1              equ     Offset
Offset          set     Offset+4
                endm


APTR            macro
\1              equ     Offset
Offset          set     Offset+4 ;E anche il Puntatore A è lungo
                endm


LABEL           macro
\1              equ     Offset
                endm
```

Fig. 11.2: Alcune Macro come in "exec/types.i"

11.3 Creazione di strutture con macro

Se osserviamo i nomi della Fig. 11.2 e abbiamo già dato un'occhiata alla documentazione Amiga, sicuramente ci troviamo in presenza di qualcosa di già noto. Si tratta di tipi di dati, come vengono utilizzati nel C per Amiga. Nell'Assembler tuttavia, non abbiamo bisogno di occuparci di determinate ricercatezze del compilatore in C. Un byte per noi è e resta un byte. Il fatto che esso in seguito debba contenere una cifra preceduta da un segno oppure no, per noi è identico. Di conseguenza non abbiamo bisogno di fare distinzione fra UBYTE (Unsigned Byte) e BYTE. Lo stesso vale per ULONG e LONG.

Nell'Assembler dell'HiSoft la questione è molto più semplice, dal momento che esso conosce l'istruzione RS. Quindi invece di

```
        WORD    xxx
```

si scriverà più semplicemente

```
        xxx     rs.w    1
```

Osserviamo quindi questo sviluppo di macro. Abbiamo per es.:

```
STRUCTURE        macro            ;Sistema per alcuni Assembler
\1               set      0       ;per dare un nome
Offset           set      \2      ;e fare in modo che
                                  ;cominci da 0
                 endm
```

Richiamiamo questa macro con

```
STRUCTURE NewWindow,0
```

Da cui deriva

```
NewWindow        set       0
Offset           set       0
```

Sono nate due costanti, e ambedue hanno il valore 0. Ora io chiamo di nuovo una macro, ma questa volta la seguente:

```
WORD             macro
\1               equ      Offset
Offset           set      Offset+2         ;Una parola ha 2 Byte
                 endm
```

La chiamata ha luogo con

```
        WORD nw_LeftEdge
```

Da cui deriva

```
nw_LeftEdge      equ      Offset
Offset           set      Offset+2
```

Conseguenza dell'esercizio: "nw _LeftEdge" ha ora il valore 0 (il vecchio valore di Offset), e l'Offset stesso si trova ora a 2. Ora, possiamo procedere nello stesso modo. L'esercizio completo è mostrato in Fig. 11.3.

In questa maniera abbiamo approntato una tabella di costanti. Se confrontiamo tale tabella con la Fig. 11.1 noteremo immediatamente che in questo caso sono stati utilizzati dei nomi simbolici per i singoli elementi. Inoltre dovrebbe essere chiaro che gli Offset corrispondono ai tipi, per cui in questo caso per una parola lunga vengono definiti per es. 4 Byte.

Incontreremo molto spesso nei file Include tali tabelle di Offset, ed in particolare in "Intuition". Tuttavia non si tratta delle strutture di cui stiamo parlando.

Istruzione	Valore di Label	Offset
STRUCTURE NewWindow,0	0	0
WORD nw_LeftEdge	0	2
WORD nw_TopEdge	2	4
WORD nw_Width	4	6
WORD nw_Height	6	8
BYTE nw_DetailPen	8	9
BYTE nw_BlockPen	9	10
ULONG nw_IDCMPFlag	10	14
LONG nw_Flags	14	18
APTR nw_FirstGadget	18	22
APTR nw_CheckMark	22	26
APTR nw_Title	26	30
APTR nw_Screen	30	34
APTR nw_BitMap	34	38
WORD nw_MinWidth	38	40
WORD nw_MinHeight	40	42
WORD nw_MaxWidth	42	44
WORD nw_MaxHeight	44	46
WORD nw_Type	46	48
LABEL nw_SIZE	48	48

Fig. 11.3: Una tabella di Offset

Una tabella di Offset non è una struttura di dati!

Anche se nei file Include della Metacomco tutto comincia con la parola STRUCTURE, non si tratta di strutture. Questo nome di macro STRUCTURE è stato scelto impropriamente. Nelle strutture dei dati è possibile scrivere qualcosa, mentre in un Offset (praticamente una costante) non è possibile. Osserviamo quindi la Fig. 11.4.

```
* window2

* In questo file sono presenti diverse dichiarazioni e Macro.
* Osserviamo attentamente!

         incdir  ":include/"
```

```
        include  intuition/intuition.i
        include  intuition/intuition_lib.i
        include  exec/exec_lib.i
        include  graphics/graphics_lib.i


* Apertura della Intuition Library :
* --------------------------
         lea     intname,a1
         moveq   #0,d0
         CALLEXEC OpenLibrary
         tst.l   d0
         beq     abbruch
         move.l  d0,_IntuitionBase         ;Assicurare il puntatore di Base

* Apertura della Ggraphics Library
* -------------------------
         lea     grafname,a1
         moveq   #0,d0
        CALLEXEC OpenLibrary
        tst.l    d0
        beq      closeint
        move.l   d0,_GfxBase               ;Assicurare il puntatore di Base


* Apertura della Window
* --------------
         jsr      InitWindow               ;inizializzazione NewWindow

         lea     NewWindow,a0              ;punta alla struttura di Window
         CALLINT OpenWindow                ;apre Window
         tst.l   d0                        ;qualcosa non ha funzionato?
         beq     closegraf                 ;se si
         move.l  d0,windowptr              ;Assicurare puntatore  Window


* Scrittura del testo nella finestra
* ------------------------
        moveq    #100,d0                   ;Posizione X
        moveq    #50,d1                    ;Y
        move.l   windowptr,a1              ;Via puntatore alla Window
        move.l   wd_RPort(a1),a1           ;Prelevamento indirizzo Rast-Port
        CALLGRAF Move                      ;Funzione Move to X,Y

        move.l   windowptr,a1              ;serve di nuovo la Rastport
        move.l   wd_RPort(a1),a1
        lea      msg,a0                    ;indirizzo testo
        moveq    #msglen,d0                ;sua  Lunghezza
        CALLGRAF Text                      ;e output

* Attesa di un Even (puo' essere solo WINDOWCLOSE)
* -------------------------------------------------
        move.l   windowptr,a0              ;Punta alla struttura Window
        move.l   wd_UserPort(a0),a0        ;ora alla  Message-Port
        move.b   MP_SIGBIT(a0),d1          ;Numereo Bit Segnale -> d1
```

```
        moveq.l  #1,d0                      ;in Maschera
        lsl.l    d1,d0                      ;       conversione
        CALLEXEC Wait                       ;Attesa!


* Chiusura della finestra
* -------------------
        move.l   windowptr,a0               ;risveglio
        CALLINT  CloseWindow                ;Chiusura finestra


* Chiusura delle Libraries
* ---------------------
closegraf
        move.l   _GfxBase,a1
        CALLEXEC CloseLibrary

closeint
        move.l   _IntuitionBase,a1
        CALLEXEC CloseLibrary

abbruch
        move.l   #0,d0                      ;oppure fine normale
        rts

W_Gadgets equ   WINDOWSIZING!WINDOWDRAG!WINDOWDEPTH!WINDOWCLOSE
W_Extras  equ   SMART_REFRESH!ACTIVATE

W_Title dc.b    'Titolo Finestra',0

**********************************************************************
NewWindow        ds.b     nw_SIZE  ;Buffer per  Struttura della  Window *
**********************************************************************

InitWindow
        lea      NewWindow,a0                ;Riempimento struttura
        move.w   #200,nw_LeftEdge(a0)
        move.w   #50,nw_TopEdge(a0)
        move.w   #300,nw_Width(a0)
        move.w   #100,nw_Height(a0)
        move.b   #0,nw_DetailPen(a0)
        move.b   #1,nw_BlockPen(a0)
        move.l   #W_Title,nw_Title(a0)
        move.l   #W_Gadgets!W_Extras,nw_Flags(a0)
        move.l   #CLOSEWINDOW,nw_IDCMPFlags(a0)
        clr.l    nw_FirstGadget(a0)
        clr.l    nw_CheckMark(a0)
        clr.l    nw_Screen(a0)
        clr.l    nw_BitMap(a0)
        move.w   #100,nw_MinWidth(a0)
        move.w   #20,nw_MinHeight(a0)
        move.w   #640,nw_MaxWidth(a0)
        move.w   #200,nw_MaxHeight(a0)
        move.w   #WBENCHSCREEN,nw_Type(a0)
        rts
```

```
intname          INTNAME           ;Nome della Intuition Lib (via Macro)
grafname         GRAFNAME          ;Nome della Graphics Lib


msg              dc.b     'Hello, World! '
msglen           equ      *-msg

_IntuitionBase   ds.l     1          ;Memoria per i puntatori
_GfxBase         ds.l     1
windowptr        ds.l     1
```

Fig. 11.4: utilizzo delle tabelle di Offset

11.4 utilizzo delle tabelle di offset

Il listato corrisponde essenzialmente a quello di Fig. 8.1, e il suo risultato é assolutamente identico. Nel paragrafo "apertura di Window" è stata aggiunta la seguente riga:

```
      jsr        InitWindow              ;Inizializzazione di NewWindow
```

e si tratta di una novità. Per aprire una Window abbiamo bisogno di una struttura con valori come quelli illustrati in Fig. 11.1. Tale struttura deve trovarsi nella RAM, per cui abbiamo bisogno di un campo di memoria. Tale Buffer viene creato con

```
    NewWindow       ds.b  nw_SIZE          ;Buffer per struttura di Window
```

con ciò avremo un Buffer vuoto con una dimensione di nw_SIZE (48 Byte). Ecco perché in Fig. 11.2 è stata determinata la dimensione. A questo punto dobbiamo riempire tale buffer con dei dati. Dal momento però che abbiamo a disposizione solo degli offset, impostiamo il registro a0 all'inizio del buffer con

```
        lea      NewWindow,a0
```

ed utilizziamo il tipo di indirizzamento, già spesso utilizzato, "ARI con Offset" per comandi come

```
        move.w   #200,nw_LeftEdge(a0)
        move.w   #50,nw_TopEdge(a0)
        move.w   #300,nw_Width(a0)
        move.w   #100,nw_Height(a0)
```

Con ciò avremmo determinato l'angolo superiore sinistro, nonché la larghezza e la lunghezza della finestra. Tuttavia non mi piace quello che abbiamo fatto, infatti avremmo ottenuto lo stesso scopo anche solo con un semplice

```
        dc.w    200,50,300,100
```

Ulteriori istruzioni dc avrebbero inizializzato il buffer esattamente come gli altri comandi di Move. In conclusione, in ambedue i casi sono disponibili nel Buffer i 48 Byte giusti. Ne abbiamo bisogno, tuttavia ora abbiamo anche disponibili i comandi Move.
Tali comandi naturalmente necessitano di memoria, in questo caso 18 x 6 = 108 Byte, e (ancora peggio) di tempo di calcolo.

Ciò si verifica quando si traducono semplicemente delle strutture in C in Assembler.
Qualcuno potrà evidenziare che questo procedimento ha altri vantaggi. Sappiamo che la struttura di NewWindow dopo l'apertura della finestra non viene più utilizzata, dal momento che Intuition crea da essa una struttura propria maggiore. Possiamo quindi riempire la struttura di NewWindow con altri dati ed utilizzarla per una seconda finestra. Per fare ciò però, abbiamo bisogno di variabili, e le avremmo solo con questa costruzione. E' chiaro? In C si, in Assembler no. Cosa ci impedisce di scrivere quanto rappresentato in Fig. 11.5?

```
W_Title_1 dc.b 'Titolo-Finestra 1',0
W_Title_2 dc.b 'Titolo-Finestra 2',0

windowdef
pos     dc.w    200,50                  ;a sinistra in alto
        dc.w    300,100                 ;Larghezza, Altezza
        dc.b    -1,-1                   ;Pen dello Screen
        dc.l    CLOSEWINDOW             ;IDCMP Flag unico
        dc.l    W_Gadgets!W_Extras      ;Flag di Window
        dc.l    0                       ;nessun User-Gadgets
        dc.l    0                       ;nessun User-Checkmark
titolo  dc.l    W_Title                 ;Titolo della Window
        dc.l    0                       ;nessun Screen proprio
        dc.l    0                       ;nessun Super Bitmap
        dc.w    100,20                  ;Dimensione Min.
        dc.w    640,200                 ;Max.
        dc.w    WBENCHSCREEN            ;Usa il Workbench Screen
```

Fig. 11.5: Con la label le strutture dc diventano flessibili

Nulla ce lo impedisce. “dc” significa infatti “definisci costanti” ma non dobbiamo prenderlo troppo alla lettera. Infatti possiamo modificare in seguito i valori. Quindi possiamo aprire una finestra come sempre. Se dopo ciò ne abbiamo bisogno di una seconda, dovrà trovarsi da un'altra parte ed avere un altro titolo. Quindi scriviamo:

```
move     #300,pos
move     #100,pos+2
move.l   #W_Title_2,titolo
```

Se vogliamo modificare ulteriormente, prevediamo semplicemente un paio di Label in più. Diversamenie, sarà sufficiente contare. Quando il primo valore ha la label “pos” e “pos” occupa 2 Byte, il valore successivo comincerà con “pos + 2”. A questo punto possiamo richiamare di nuovo OpenWindow, nel modo che segue:

```
lea      NewWindow,a0     ;punta alla struttura di Window
CALLINT OpenWindow        ;apre Window
tst.l    d0               ;qualcosa non ha funzionato?
beq      close_1          ;se si
move.l   d0,windowptr_2   ;Assicurare il puntatore della Window
```

Facciamo attenzione alla piccola differenza. Abbiamo una nuova Window quindi dobbiamo memorizzare naturalmente il suo puntatore in una propria variabile, in questo caso “windowptr_2”.

Un'altra differenza: se all'apertura della Window 2 qualcosa non ha funzionato, non si potrà saltare a “closeint”, bensì alla riga a partire dalla quale viene chiusa la Window 1. E’ immediatamente dopo di essa che dovrebbe esserci “closeint”.

Se si prova con più window (ed è un esercizio che vi consiglio) sarà naturalmente possibile aprire una finestra dopo l'altra e chiamare nel blocco successivo “Waits” con i relativi puntatori alla window. In questo caso sarà comunque possibile chiudere le finestre solo nella stessa sequenza. Una soluzione più pratica è offerta dal “GetMsg/RepIyMsg” di Capitolo 9. Ultimo consiglio: sarà opportuno annotare in variabili extra se una finestra è aperta o chiusa. Infatti non sarà possibile rivolgersi ad una finestra che è già chiusa.

In conclusione, un consiglio positivo per le tabelle di Offset. Come già detto, Intuition crea dalla nostra struttura di NewWindow una struttura maggiore nella RAM. Al fine di poter accedere ad elementi di questa struttura, è necessario naturalmente conoscere gli Offset. Li abbiamo già utilizzati, ecco un esempio:

```
move.l   windowptr,a0       ;Punta alla struttura di Window
move.l   wd_UserPort(a0),a0 ;ora alla Message-Port
```

Ricordate? In questa struttura (ed in altre dello stesso genere) è contenuto molto di più. Ecco perché ce ne occuperemo, dopo questo breve corso, nel capitolo successivo in maniera più approfondita.

11.5 BPTR e BSTR

Nelle macro dei file Include o in altre fonti, ci scontreremo spesso con i concetti BPTR e BSTR. Si tratta di un puntatore BCPL oppure di una stringa BCPL. BCPL è un linguaggio simile al C, nel quale il Software Amiga è stato parzialmente sviluppato. Dal momento che BCPL è stato concepito per computer che lavorano con indirizzamento di parola lunga, abbiamo un problema, in quanto, come noto, il 68000 è una macchina a Byte, cioé può indirizzare ogni Byte, mentre BCPL lo può fare solo per 1 ogni 4.

Di conseguenza un BPTR deve sempre puntare ad un limite di parola lunga. Se troviamo questi puntatori in strutture di dati, tale struttura deve venire giustificata a “lungo”, ottenibile con “cnop 0,4” (SEKA: align 4). Lo stesso vale per BSTR. BSTR punta al primo byte di una stringa. Tale byte contiene la lunghezza della stringa, i byte seguenti sono il testo vero e proprio.

Nel caso in cui vogliamo utilizzare tali puntatori, dovremo effettuare dei calcoli, cioè moltiplicare per 4 il BTR. In Assembler ciò ha luogo molto velocemente tramite uno spostamento a sinistra di 2.

Un APTR è d’altra parte un puntatore comunissimo (un indirizzo), che verrà utilizzato come parola lunga, cioè esattamente com'è.

CAPITOLO 12

Intuition completo

Screen
Window
Font
Event
Menu
Gadget

12.1 Screen

Con l'Amiga è possibile aprire un numero di Screen a piacere. Diciamo meglio, quasi a piacere, dal momento che uno Screen necessita anche di memoria.

Uno Screen è un cosiddetto schermo virtuale, cioè uno schermo apparente, il quale tuttavia si comporta in linea di massima come uno schermo vero. Esso potrà quindi essere a colori oppure monocromatico, avere diverse dimensioni, diverse risoluzioni ed altre caratteristiche identificative. E' inoltre possibile rappresentare più di uno di questi Screen contemporaneamente sullo schermo Hardware.

Contrariamente alle window, questi Screen non possono sovrapporsi parzialmente, ma possono solo trovarsi l'uno sull'altro (non l'uno vicino all'altro) e con il Mouse potranno venire spostati solo verticalmente.

Uno Screen può contenere diverse window, influenzandone le caratteristiche.

Apertura di uno Screen

Normalmente uno Screen viene aperto esattamente come una window. Serve una struttura (in questo caso NewScreen) un puntatore a questa struttura ed una funzione, la quale naturalmente si chiama _LVOOpenScreen. Come risultato otteniamo un puntatore ad una struttura, che è elencato e spiegato in appendice. E' importante annotare questo indirizzo ed è importantissimo immettere tale indirizzo anche nella window (in tutte le window). La routine di base per quanto concerne lo Screen è:

```
lea      NeuScreen(pc),a0 ;Puntatore alla struttura
CALLINT  OpenScreen       ;Apertura Screen
tst.l    d0               ;qualcosa non va?
beq      closegfx         ;se si
move.l   d0,Screen        ;memorizzazione nella Window!
```

L'ulteriore inizio di programma di Fig. 12.1 è già noto. Osserviamo tuttavia due modifiche nella struttura di window: una è data dal fatto che la Label Screen è stata aggiunta, e che in questo punto viene scritto l'indirizzo della struttura di Screen, l'altra è che è stato modificato l'input WBENCHSCREEN in CUSTOMSCREEN. Se apriamo una seconda finestra, che per es. ha come label Screen_1, dobbiamo prima di tutto introdurre l’indirizzo di Screen con

```
move.l   Screen,Screen_1
```

12.2 Font

A scopi dimostrativi (e anche per prepararvi al presente paragrafo) ho già impostato un altro Font quando ne ho avuto l'occasione. Dallo Screen, un puntatore punta a Font. In questa struttura un puntatore punta di nuovo al nome di Font. Proviamo pure tranquillamente altri Font e dimensioni (ved. Appendice). L'importante è il seguente principio: "il puntatore punta alla struttura, il cui puntatore punta all'altra struttura".

```
        opt     l-

*screen.s

        incdir  ":include/"
        include exec/exec_lib.i
        include intuition/intuition.i
        include intuition/intuition_lib.i
        include graphics/graphics_lib.i
        include graphics/text.i

        moveq   #0,d0                   ;Apertura di Intuition
        lea     int_name(pc),a1
        CALLEXEC OpenLibrary
        tst.l   d0
        beq     abbruch
        move.l  d0,_IntuitionBase

        moveq   #0,d0                   ;Apertura di Graphics
        lea     graf_name(pc),a1
        CALLEXEC OpenLibrary
        tst.l   d0
        beq     closeint
        move.l  d0,_GfxBase

        lea     NewScreen(pc),a0
        CALLINT OpenScreen              ;Open Screen
        tst.l   d0
        beq     closegfx
        move.l  d0,Screen               ;memorizzazione nella Window!

        lea     NewWindow(pc),a0        ;Open Window
        CALLINT OpenWindow
        tst.l   d0
        beq     closescr
        move.l  d0,Window

        move.l  d0,a1                   ;output di testo
        move.l  wd_RPort(a1),a1         ;Move
        moveq   #100,d0
```

```
        moveq    #50,d1
        CALLGRAF Move

        move.l   Window(pc),a0             ;print
        move.l   wd_RPort(a0),a1
        lea      msg(pc),a0
        moveq    #msglen,d0
        CALLGRAF Text

        move.l   Window(pc),a0             ;attesa di Event
        move.l   wd_UserPort(a0),a0
        move.b   MP_SIGBIT(a0),d1
        moveq    #0,d0
        bset     d1,d0
        CALLEXEC Wait

        move.l   Window(pc),a0             ;close all
        CALLINT CloseWindow

closescr
        move.l   Screen(pc),a0
        CALLINT CloseScreen

closegfx
        move.l   _GfxBase(pc),a1
        CALLEXEC CloseLibrary

closeint
        move.l   _IntuitionBase(pc),a1
        CALLEXEC CloseLibrary

abbruch
        rts


NewScreen        dc.w     0,0               ;left, top
                 dc.w     320,200           ;width, height
                 dc.w     2                 ;depth
                 dc.b     0,1               ;pens
                 dc.w     0                 ;viewmodes
                 dc.w     CUSTOMSCREEN      ;type
                 dc.l     Font              ;font
                 dc.l     screen_title      ;title
                 dc.l     0                 ;gadgets
                 dc.l     0                 ;bitmap
Font
                 dc.l     font_name
                 dc.w     TOPAZ_SIXTY
                 dc.b     FS_NORMAL
                 dc.b     FPF_ROMFONT

W_Gadgets equ   WINDOWDRAG!WINDOWDEPTH!WINDOWCLOSE
```

```
W_Extras  equ   SMART_REFRESH!ACTIVATE
W_Title dc.b    'Titolo Finestra',0
        cnop     0,2

NewWindow
       dc.w     20,20                   ;a sinistra in alto
        dc.w    300,100                 ;Larghezza , Altezza
        dc.b    0,1                     ;Pen dello Screen
        dc.l    CLOSEWINDOW
        dc.l    W_Gadgets!W_Extras      ;Flag della Window
        dc.l    0                       ;nessun User-Gadget
        dc.l    0                       ;nessun User-Checkmark
        dc.l    W_Title                 ;Titolo della Window
Screen  ds.l    1                               ;Screen proprio
        dc.l    0                       ;nessuna Super Bitmap
        dc.w    100,20,640,200          ;Limiti di dimensione della
Window
        dc.w    CUSTOMSCREEN            ;Usa nostro screen

_IntuitionBase  dc.l    0
_GfxBase dc.l   0
Window          dc.l    0


int_name INTNAME
graf_name       GRAFNAME
msg             dc.b    'Hello Amiga'
msglen          equ      *-msg
                cnop    0,2
screen_title    dc.b    'Unser Screen',0
font_name       dc.b    'topaz.font',0
```

Fig. 12.1: Screen e testo propri

12.3 Event

Con la Fig. 12.2, siamo arrivati all'argomento “vari”, vediamo quindi subito l’argomento “Event”.

```
* event.s

* In questi file sono contenute diverse Dichiarazioni e Macro.
* Osserviamo attentamente!

         incdir  ":include/"
```

```
        include  intuition/intuition.i
        include  intuition/intuition_lib.i
        include  exec/exec_lib.i
        include  graphics/graphics_lib.i

* Scrittura del testo nella finestra
* ------------------------
PRINT            macro
                 moveq    \1,d0              ;Posizione X
                 moveq    \2,d1              ;Y
                 move.l   windowptr,a1       ;Via Puntatore alla Window
                 move.l   wd_RPort(a1),a1    ;Prelevamento indirizzo Rast-Port
                 CALLGRAF Move               ;Funzione Move to X,Y
                 move.l   windowptr,a1       ;serve di nuovo la Rastport
                 move.l   wd_RPort(a1),a1
                 lea      \3,a0               ;indirizzo testo
                 moveq    \4,d0              ;sua lunghezza
                 CALLGRAF Text               ;ed ouput
                 endm


_main

* Apertura della Intuition Library
* --------------------------
        lea      intname,a1
        moveq    #0,d0
        CALLEXEC OpenLibrary
        tst.l    d0
        beq      abbruch
        move.l   d0,_IntuitionBase          ;Assicurare il Puntatore di base

* Apertura della Graphics Library
* ------------------------
        lea      grafname,a1
        moveq    #0,d0
        CALLEXEC OpenLibrary
        tst.l    d0
        beq      closeint
        move.l   d0,_GfxBase                ;Assicurare il Puntatore di base

* Apertura della Window
* --------------
        lea      windowdef,a0               ;Punta alla struttura di Window
        CALLINT  OpenWindow                 ;apre la Window
        tst.l    d0                         ;qualcosa non ha funzionato?
        beq      closegraf                  ;se si
        move.l   d0,windowptr               ;Assicurare il puntatore della
Window

* Installazione del Menu
* -----------------
        move.l   d0,a0                      ;^Window
        lea      Menu0,a1                   ;^Menu
```

```
        CALLINT  SetMenuStrip             ;do it


* Attesa di un Event, quindi valutazione
* --------------------------------
event
        move.l   windowptr,a0             ;Punta alla struttura di Window
        move.l   wd_UserPort(a0),a0       ;ora alla Message-Port
        move.l   a0,a5                    ;salvataggio indirizzo della
Porta
        move.b   MP_SIGBIT(a0),d1         ;Prelevamento del Bit segnale
        moveq    #0,d0                    ;Trasformare Numero
        bset     d1,d0                    ;in Maschera
        CALLEXEC Wait                     ;Attesa!

        move.l   a5,a0                    ;Prelev. indirizzo della Porta
        CALLEXEC GetMsg                   ;Prelevamento del Messaggio
        move.l   d0,a1                    ;deve andare in a1
        move.l   im_Class(a1),d4          ;Tipo di Msg
        move.w   im_Code(a1),d5           ;Sottogruppo
        move.l   im_IAddress(a1),a4       ;indirizzo per Gadget
        CALLEXEC ReplyMsg                 ;muovere Msg in a1

        cmpi.l   #CLOSEWINDOW,d4          ;Window Closed?
        beq      closewindow              ;se si

        cmpi.l   #MENUPICK,d4             ;Menu scelto?
        beq      do_menu                  ;se si

        cmpi.l   #GADGETUP,d4             ;e cosi' via per
        beq      do_gadget                ;ogni Bit

        bra       event

do_menu
        cmpi     #MENUNULL,d5             ;Item selezionato?
        beq      event                    ;se no
        move     d5,d2                    ;Code
        lea       buffer+8,a0             ;visualizzazione in esa
        bsr       hex                     ;Chiamata Routine
        PRINT     #50,#50,buffer,#12
        bra       event

do_gadget
        sub.l    a0,a0
        CALLINT  DisplayBeep              ;accade qualcosa
        bra      event


closewindow
        move.l   windowptr,a0             ;risveglio
        CALLINT  CloseWindow              ;Chiusura finestra
```

```
* Chiusura Libraries
* --------------------
closegraf
        move.l   _GfxBase,a1
        CALLEXEC CloseLibrary

closeint
        move.l   _IntuitionBase,a1
        CALLEXEC CloseLibrary




abbruch
        moveq    #0,d0                    ;oppure fine normale
        rts

* Conversione di d2.w in Stringa ASCII a partire da (a0)
* ----------------------------------------
hex
        moveq    #3,d1                    ;per 4 Nibble
next    rol      #4,d2                    ;Prelevamento di 1 Nibble
        move     d2,d3                    ;salvataggio in d3
        and.b    #$0f,d3                  ;mascheratura
        add.b    #48,d3                   ;trasformazione in ASCII
        cmp.b    #58,d3                   ;e' >9 ?
        bcs      out                      ;se no
        addq.b   #7,d3                    ;altrimenti deve essere A-F
out     move.b   d3,(a0)+                 ;Memorizzazione di 1 carattere
        dbra     d1,next                  ;prossimo nibble
        rts

        buffer   dc.b     'Code =  xxxx  hex'
        cnop     0,2
gets    equ      WINDOWSIZING!WINDOWDRAG!WINDOWDEPTH!WINDOWCLOSE
W_Extras equ     SMART_REFRESH!ACTIVATE

W_Title  dc.b    'Titolo finestra',0

windowdef
        dc.w     200,50                   ;a sinistra in alto
        dc.w     300,100                  ;Larghezza, Altezza
        dc.b     -1,-1                    ;Pen dello Screen
        dc.l     CLOSEWINDOW!MENUPICK!GADGETUP!GADGETDOWN
        dc.l     W_Gadgets!W_Extras       ;Flag della Window
        dc.l     Gadget0                  ;primo  User-Gadget
        dc.l     0                        ;nessun User-Checkmark
        dc.l     W_Title                  ;Titolo della  Window
        dc.l     0                        ;nessuno Screen proprio
        dc.l     0                        ;nessuna  Super Bitmap
        dc.w     100,20                   ;Dimensione Min.
        dc.w     640,200                  ;Max.
        dc.w     WBENCHSCREEN             ;Usa Workbench Screen
```

```
Menu0
        dc.l    0                       ;niente di piu'
        dc.w    50,0                    ;x,y
        dc.w    60,0                    ;larghezza, altezza
        dc.w    MENUENABLED             ;Flags
        dc.l    MName0                  ;^Titolo
        dc.l    Item0                   ;^lista Item
        dc.w    0,0,0,0                 ;Uso sistema
Item0
        dc.l    Item1                   ;^prossimo Item
        dc.w    0,0                     ;x,y
        dc.w    100,12                  ;larghezza, altezza
        dc.w    ITEMENABLED!ITEMTEXT!HIGHCOMP!COMMSEQ  ;Flags
        dc.l    0                       ;nessun Exclude
        dc.l    IName0                  ;^Testo
        dc.l    0                       ;Select Fill
        dc.b    'N'                     ;Cmd-Key
        dc.b    0                       ;Dummy
        dc.l    0                       ;nessun Sub-Item
        dc.w    0                       ;     next Select

Item1
        dc.l    0                       ;^next Item
        dc.w    0,12                    ;x,y
        dc.w    100,12                          ;larghezza, altezza
        dc.w    ITEMENABLED!ITEMTEXT!HIGHCOMP!COMMSEQ  ;Flags
        dc.l    0                       ;nessun Excludes
        dc.l    IName1                  ;^Text
        dc.l    0                       ;Select Fill
        dc.b    'E'                     ;Cmd-Key
        dc.b    0                       ;Dummy
        dc.l    0                       ;nessun Sub-Item
        dc.w    0                       ;     next Select


IName0
        dc.b    0,2                     ;Pens
        dc.b    RP_JAM1,0               ;Modo di scrittura
        dc.w    2,2                     ;x,y
        dc.l    0                       ;System-Font
        dc.l    stri0                   ;^Text
        dc.l    0                       ;non piu' testo
IName1
        dc.b    0,2                     ;Pens
        dc.b    RP_JAM1,0               ;Modo di scrittura
        dc.w    2,2                     ;x,y
        dc.l    0                       ;System-Font
        dc.l    stri1                   ;^Testo
        dc.l    0                       ;non piu' testo

MName0  dc.b    'Menu 0',0
```

```
        cnop    0,2
stri0   dc.b    'Item 0',0
        cnop    0,2
stri1   dc.b    'Item 1',0
        cnop    0,2


* Strutture dei Gadget
* -----------------

Gadget0
        dc.l    Gadget1                         ;ancora uno
        dc.w    20,20                           ;x,y
        dc.w    40,20                           ;larghezza, altezza
        dc.w    GADGHCOMP                       ;flag
        dc.w    RELVERIFY                       ;Attivazione
        dc.w    BOOLGADGET                      ;Tipo
        dc.l    Border0                         ;^struttura Border.
        dc.l    0
        dc.l    G0text                          ;^struttura testo.
        dc.l    0,0
        dc.w    0                               ;ID
        dc.l    0
Border0
        dc.w    0,0                             ;x,y
        dc.b    15,0                            ;Pen
        dc.b    RP_JAM1
        dc.b    5                               ;Coppie
        dc.l    paar0                           ;^Lista
        dc.l    0                               ;non piu' Border
paar0
        dc.w    0,0                             ;Lista coordinate
        dc.w    0,19
        dc.w    39,19
        dc.w    39,0
        dc.w    0,0
G0text
        dc.b    7,0                             ;Pens
        dc.b    RP_JAM1
        dc.w    10,7                            ;x,y
        dc.l    0                               ;Font di sistema
        dc.l    strg0
        dc.l    0
strg0   dc.b    'G0',0
        cnop    0,2

Gadget1
        dc.l    0
        dc.w    80,20                           ;x,y
        dc.w    40,20                           ;larghezza, altezza
        dc.w    GADGHCOMP                       ;flag
        dc.w    RELVERIFY                       ;Attivazione
        dc.w    BOOLGADGET                      ;Tipo
```

```
        dc.l    Border1                 ;^struttura Border.
        dc.l    0
        dc.l    G1text                  ;^struttura testo.
        dc.l    0,0
        dc.w    0                       ;ID
        dc.l    0
Border1
        dc.w    0,0                     ;x,y
        dc.b    15,0                    ;Pen
        dc.b    RP_JAM1
        dc.b    5                       ;Coppie
        dc.l    paar1                   ;^Lista
        dc.l    0                       ;non piu' Border
paar1
        dc.w    0,0                     ;Lista coordinate
        dc.w    0,19
        dc.w    39,19
        dc.w    39,0
        dc.w    0,0
G1text
        dc.b    7,0                     ;Pen
        dc.b    RP_JAM1
        dc.w    10,7                    ;x,y
        dc.l    0                       ;Font di sistema
        dc.l    strg1
        dc.l    0
strg1   dc.b    'G1',0
        cnop    0,2

intname  INTNAME                ;Nome della Intuition Lib (via Macro)
grafname GRAFNAME               ;Nome della Graphics Lib


_IntuitionBase  ds.l    1               ;Memoria per i puntatori
_GfxBase        ds.l    1
windowptr       ds.l    1
```

Fig. 12.2: Trattamento di un Event in Intuition

Un event è un evento, e dal punto di vista del computer gli input dell'utente sono un evento. In ogni caso accade quanto segue: conosciamo già la sequenza

```
        move.l  windowptr,a0            ;Punta alla struttura di Window
        move.l  wd_UserPort(a0),a0      ;ora alla  Message-Port
        move.l  a0,a5                   ;salvataggio indirizzo della
Porta
        move.b  MP_SIGBIT(a0),dl        ;Prelevamento del bit segnale
        moveq   #0,d0                   ;Trasformare Numero
        bset    d1,d0                   ;in Maschera

        CALLEXEC Wait                   ;Attesa!
```

A questo punto il nostro Task va a dormire. Esso verrà svegliato solo quando si presenta un evento. Abbiamo determinato nella definizione della finestra come esso deve essere. Finora abbiamo sempre potuto tralasciare il fatto che l'evento si chiamasse CLOSEWINDOW. Ora invece abbiamo permesso che i seguenti Bit siano degli eventi, tramite una operazione di OR (in Assembler!):

```
dc.l CLOSEWINDOW!MENUPICK!GADGETUP!GADGETDOWN
```

A questo punto dobbiamo ancora verificare quale di questi eventi si è presentato,
Per fare ciò, dobbiamo leggere il messaggio (notizia), come segue:

```
move.l   a5,a0    ;Prelevamento indirizzo della Porta
CALLEXEC GetMsg   ;prelevamento del Message
move.l   d0,a1    ;necessariamente in a1
```

A1 punta ora alla porta messaggio, che è anch'essa una struttura. Come al solito possiamo accedere di nuovo, tramite degli Offset, agli elementi di questa struttura, dei quali ci interessano i seguenti:

```
move.l   im_Class(a1),d4          ;Tipo di Msg
move.w   im_Code(a1),d5           ;Sottogruppo
move.l   im_IAddress(a1),a4       ;Indirizzo per Gadgets
```

Il tipo corrisponde esattamente al nostro Bit di flag della Window. Sarà possibile per es. scrivere:

```
cmpi.l   #CLOSEWINDOW,d4   ;Window Closed?
beq      closewindow       ;se si
```

Il Code fornisce un sottogruppo. Se per es. il tipo MENUPICK è stato riconosciuto, possiamo andare a verificare nel Code di quale menu o di quale sottomenu si tratti.

L'indirizzo che abbiamo salvato qui in a4, sarà necessario in caso di Gadget. Tale indirizzo punta all'inizio della struttura dei gadget (il cui gadget è stato clickato). Tramite un Offset sarà quindi possibile prendere l'ID (numero del gadget):

```
move     gg_GadgetID(a4),d0
```

A questo punto, come al solito, prima di chiamare Intuition o un'altra routine di sistema, dobbiamo assolutamente abbandonare la notizia. Ciò accade con

```
CALLEXEC ReplyMsg          ;abbandonare Msg in a1
```

Inoltre non dobbiamo lasciar trascorrere molto tempo fino al momento di "Reply".
Nel nostro caso abbiamo quattro comandi intermedi, e ciò va ancora bene.

12.4 Menu

Finora con MENUPICK avevamo dato il permesso di venire svegliati in caso di selezioni da menu, ma prima bisogna fare qualcos'altro. I menu vengono installati con

```
move.l   d0,a0           ;Window
lea      Menu0,a1        ;Menu
CALLINT  SetMenuStrip    ;do it
```

Facciamo comunque attenzione che un menu è sempre collegato ad una finestra (anche se appare sempre nelle righe di schermo). Di conseguenza il puntatore alla struttura di finestra sarà sempre il primo parametro. Quindi dobbiamo attribuire un puntatore alla struttura di menu per poter, infine, installare la “striscia di menu”.

Nella determinazione della struttura avremo molto da fare. Le lunghe operazioni di battitura potranno tuttavia venire molto semplificate, scrivendo dapprima un menu e duplicando tutto il resto, modificandolo solo in alcuni punti.

Infatti per ogni titolo di menu esiste una struttura, che punta sempre a quella successiva, finché il puntatore dell’ultima non diventa zero. In ogni struttura di titolo esiste un puntatore alla lista degli Item (sottotitoli) che segue lo stesso principio. In ogni struttura di Item esiste un puntatore ad una struttura di testo, ed in essa di nuovo un puntatore al testo vero e proprio.

Nella Routine do_menu il codice viene rappresentato semplicemente come parola in esadecimale. I due menu non bastano per la soluzione dell'indovinello, per cui vediamola qui. Nella parola a 16 Bit di Code sono contenute tre cifre, cioè i numeri per il titolo, l'ltem ed il Sub-Item. Per quest'ultimo, vengono memorizzati negli Item dei puntatori a delle strutture, le quali si comportano come gli Item. Il codice viene risolto quindi con

Bit   0 - 4: Titolo
Bit   5 -10: Item
Bit 11 -15: Sub-ltem

I numeri vengono contati sempre a partire da 0. Normalmente si copierà il registro, per il titolo si effettuerà una operazione di AND con la parola contenente #%11111, ottenendo così il numero di titolo. Nella routine di titolo si prenderà di nuovo il codice originale, lo si sposterà verso destra di 5 Bit, si effettuerà una operazione di AND con %#111111 e si otterrà l'Item. Se per determinati Item erano previsti anche dei Sub-ltem, si continuerà nella stessa maniera (spostamento di 11 e operazione di AND con #%11111). Facciamo inoltre attenzione che tramite una costante è possibile spostare solo fino ad un massimo di 8, per spostare 11 saranno quindi necessari due comandi (uno con 8 e uno con 3). In alternativa si caricherà un registro con 11 e si effettuerà lo spostamento tramite esso.

12.5 Gadget

Abbiamo già visto come ottenere un ID di gadget. tunavia ci mancano ancora due premesse. Nella finestra, “User-Gadget” dovrà puntare alla prima struttura di gadget, e quest'ultima dovrà naturalmente esistere.

Ricominciamo con le “puntature”. Un gadget punta a quello successivo. In un gadget c'è un puntatore ad una struttura di bordo (chiamato poligono). in esso c'è un altro puntatore ad una lista,contenente le coppie di coordinate per ogni angolo del poligono.
Accontentiamoci qui di un semplice quadrato. Quindi c'è un altro puntatore alla struttura di testo già nota, dalla quale un ulteriore puntatore punta alla stringa di testo.

12.6 Requester

Potremmo ora produrre moltissimi punti di menu e gadget da riempire intere pagine, ma ci annoieremmo molto. Anche altri tipi di strutture non sono più difficili. Tutto ciò è contenuto nel “Intuition Reference Manual” di cui vi consiglio la lettura. Per esercizio dovremmo provare ad incorporare nel programma un Requester, la cui chiamata si trovi per es. nella posizione nella quale facciamo lampeggiare brevemente in rosso lo schermo con “DisplayBeep”.

Il caso più semplice è l'Auto-Requester, che normalmente pone una domanda e permette due risposte (due gadget). La chiamata ha luogo con

```
        CALLINT  AutoRequest
```

ma prima dovrà essere stato caricato quanto segue:

a0: puntatore alla struttura di finestra (in questo caso lea windowptr,a0)
a1: puntatore alla struttura di testo con la domanda (prendiamo semplicemente lea MName0,a1)
a2: puntatore alla struttura di testo con testo, gadget sinistro
a3: puntatore alla struttura di testo con testo, gadget destro
(prendiamo semplicemente lea G0text,a2
lea G1text,a3)
d0,d1: Flag IDCMP, che indicano cosa deve venire riconosciuto come per es. un clickaggio nel gadget sinistro/destro
(prendiamo dapprima clr.l d0 / clr.l d1)
d2.l,d3.l: larghezza ed altezza del Requester

12.7 Trasposizione da C in Assembler

Nel manuale Hardware troveremo molti esempi per la programmazione diretta dell'Hardware per la grafica in Assembler. Se si è interessati ad una grafica estremamente veloce, è consigliabile l'acquisto di tali strumenti. Diversamente, la maggior parte degli esempi della documentazione per l'Amiga sono scritti in C, anche Intuition, e ciò crea un motivo sufficiente affinché noi ci occupiamo, almeno per una volta, della traduzione in Assembler.

Un “#include” viene sostituito con “include”; per quanto concerne i nomi di file,si modificheranno le estensioni da h in i. Per

```
#define Nome Valore
```

scriveremo

```
Nome equ Valore
```

I nomi delle funzioni coincidono gli uni con gli altri, è tuttavia necessario preporre sempre _LVO. Se passiamo all'esame delle macro, ci avviciniamo sempre di più al C, in quanto per es.

```
move.l   _IntuitionBase,a6
jsr      _LVOOpenWindow(a6)
```

tramite una macro diventerà semplicemente

```
CALLINT OpenWindow
```

I tipi di dati in C fanno “molto rumore per nulla”, per cui vale:

Tipo C	Tipo Assembler
int	.W
long int	.L
unsigned int	.W
char	.B
BYTE	.B
UBYTE	.B
WORD	.W
UWORD	.W
LONG	.L
ULONG	.L

I tipi scritti con lettere maiuscole non sono dei veri e propri tipi per C, ma solo delle macro. L'istruzione usuale in C per una chiamata

```
if OpenWindow(&windowptr)==0
     exit(false)
```

viene sostituita con

```
tst.l d0
beq exit          ;là Close Lib ecc. e rts
```

Una espressione del tipo

```
windowprt->wd_RPort
```

diventerà in Assembler

```
move.l windowptr,a1
move.l wd_RPort(a1),a1
```

Le strutture possono venire riconosciute sulla base dei loro nomi. Talvolta troveremo i loro nomi con il prefisso “extern”. Si tratta di una istruzione al Linker di collegare tale struttura.

In conclusione: l'Assembler è sempre più veloce, non solo perché il codice che scrivere-mo è migliore (tagliato su misura) di quello del compilatore in C, bensì anche per un altro semplice motivo. L'attribuzione dei parametri alle routine di sistema avviene sempre tramite registri, come abbiamo già visto in precedenza. In C essa avviene tramite definizione del linguaggio attraverso lo Stack. Di conseguenza il compilatore in C deve produrre un codice che dapprima metta i parametri nello Stack e quindi produca un co-siddetto Binding (collegamento) che vada a prendere tali parametri dallo Stack e li carichi nei registri.

CAPITOLO 13

Collegamento di routine di assembler in BASIC

Esigenze delle routine
Spazio per le routine
Caricamento e chiamata
Trasferimento dei parametri
Chiamata di comandi CLI in BASIC

13.1 Esigenze delle routine

Indipendenza dalla Posizione

Nell'Amiga non c'è nessun posto sicuro, come per esempio potrebbe venire definita la parte di RAM video non utilizzata dell'Atari ST. D'altra parte questa tecnica non è consigliabile, in quanto, se veramente ci fosse una zona di memoria non occupata nell'Amiga, tutte le routine di assembler vorrebbero memorizzare in essa, cosa che condurrebbe a conflitti.

Da ciò deriva la prima esigenza delle routine, cioè l'essere indipendenti dalla posizione: position independent. Ciò rende impossibile purtroppo indirizzamenti quali

```
        move.l   #buffer,d2
```

Un aiuto ci viene offerto dall'indirizzamento relativo al PC (vedi anche Capitolo 3.6). Naturalmente non avrebbe senso scrivere qualcosa come "move.l buffer(pc),d2", in quanto "buffer" è un indirizzo e non una distanza di indirizzamento. E' vero che tale distanza potrebbe venire calcolata, tuttavia è molto più semplice la forma seguente:

```
        lea      buffer(pc),a0
        move.l   a0,d2
```

A questo punto resterebbe ancora un problema: l'indirizzamento relativo al PC non è possibile per operandi di destinazione. Infatti è vietato scrivere quanto segue:

```
        move     #1,verso(pc) ;sbagliato!
```

Anche in questo caso possono esserci d'aiuto due comandi, cioè:

```
        lea      verso(pc),a0
        move     #1,(a0)
```

Al fine di scrivere per esempio in una struttura di Window, si può anche procedere come segue:

```
        lea      windowdef(pc),a0
        move     #1,4(a0)
```

Ciò sarebbe l'alternativa giusta per

```
        move     #1,windowdef+4
```

Un Solo Segmento

Per la memoria delle variabili, dei dati e dei programmi è possibile creare un solo segmento. Ciò non è un problema, basterà solo tralasciare istruzioni quali SECTION, DATA e BSS. Il motivo è che l'indirizzamento relativo al PC permette solo una distanza di 16 bit preceduta da un segno (+/- 32 Kbyte), mentre il caricatore può posizionare i segmenti a distanze maggiori. E’ quindi logico che anche il programma non possa superare i 32 Kbyte, o che per lo meno i comandi e gli indirizzi ad essi relativi non distino tra loro di più di 32 Kbyte.

Il Sottoprogramma Deve Salvare Tutti i Registri

Il fatto che la routine debba terminare con RTS é chiaro. Infatti, alla fine, essa dovrà ritornare al BASIC. E’ importante invece salvare all’inizio della routine tutti i registri (movem) e rimemorizzare prima di RTS.

Memorizzazione Solo di Code e Dati

Sarà necessario assemblare le routine di assembler, tuttavia non linkarle. Il linkaggio infatti aggiunge diverse informazioni per il caricatore (del DOS), che non sono necessarie nel BASIC. Se l'Assembler non è in grado di memorizzare dei puri moduli oggetto, noi stessi dovremo eliminare l'“Overhead”. Troveremo in seguito un esempio per questo aspetto.

13.2 Spazio per le routine

Lo Spazio per le routine deve venire preparato dal programma in BASIC. Per fare ciò è possibile definire in BASIC una matrice sufficientemente grande o, più semplicemente, utilizzare una stringa, che diventi suffcientemente grande da sola, quando si carica il code in essa (vedremo in seguito come).

13.3 Caricamento e chiamata di routine in Assembler

In Figura 13.1 è rappresentata la routine di assembler con la quale vogliamo cominciare. Il programma apre una finestra, aspetta la pressione di un tasto, ed è pronto.
In primo luogo dobbiamo occuparci della rappresentazione di come si accede a campi di dati indipendentemente dalla posizione.

```
        opt     l+,p+                   ;linkabile e pos. indep.
*b1.s
        include dos_equates             ;_LVO-Equates!!!

RELO    macro
        lea     \1(pc),a0
        move.l  a0,\2
        endm
_main
        movem.l d0-d6/a0-a6,-(sp)       ; Salvare per il Basic
        lea     dosname(pc),a1          ;Nome di DOS-Lib
        moveq   #0,d0                   ;Version indifferente
        move.l  _SysBase,a6             ;Base di Exec
        jsr      _LVOOpenLibrary(a6)    ;Apertura di DOS-Lib
        tst.l   d0                      ;errore?
        beq     fini                    ;se errore, fine
        move.l  d0,a6                   ;Annotare il puntatore

        RELO    name,d1
        move.l  #1005,d2                ;Status = c'e'
        jsr     _LVOOpen(a6)            ;Ora apertura
        move.l  d0,d5                   ;Annotare Handle
        tst.l   d0                      ;errore?
        beq     fini                    ;se si, interruzione

        move.l  d5,d1                   ;Lettura da CON
        RELO    buffer,d2               ;in questo buffer
        move.l  #1,d3                   ;1 carattere
        jsr     _LVORead(a6)            ;Chiamata di lettura

        move.l  d5,d1                   ;Chiusura di CON
        jsr     _LVOClose(a6)

        move.l  a6,a1                   ;Base di DOS-Lib
        move.l  _SysBase,a6             ;Base di Exec
        jsr     _LVOCloseLibrary(a6)    ;Funzione "Chiusura"
        movem.l (sp)+,d0-d6/a0-a6

fini    rts                             ;Ritorno al CLI

dosname dc.b    'dos.library',0
        cnop    0,2
name    dc.b    'CON:40/100/580/80/hit any key',0
        cnop    0,2
buffer  ds.b    2
```

Fig. 13.1: Programma 1 Che deve venire chiamato dal BASIC

Con una Macro RELO abbiamo un po' razionalizzato il “position independent”. Tale metodo porta dei vantaggi anche in programmi che non devono venire chiamati dal BASIC. Infatti, con esso, nel code non sono più necessari gli Offset di relocazione, quindi il codice diventa più corto e può venire caricato più velocemente. Anche se si rinuncia a RELO e si utilizza, dove è possibile, un “(PC)”, è già qualcosa. Il “compiler Switch” “p+” (DEVPAC) d'altra parte, è molto utile. Infatti esso non produce dei code dipendenti dalla posizione, ma segnala come errori le righe che dipendono dalla posizione.

Se un programma in BASIC deve chiamare una routine di Assembler, questa deve naturalmente trovarsi nella RAM. Il metodo più semplice è caricare il file binario con il code in esso. Ciò tuttavia per l'utente è piuttosto oneroso, in quanto questi deve sempre stare pronto, oltre che per il programma in BASIC, anche per il file code relativo.
Tuttavia il procedimento deve venire presentato, in quanto (in fase di test) è semplice e veloce. La figura 13.2 mostra la soluzione.

```
OPEN":libro/a" AS 1
l=LOF(1)
CLOSE 1

OPEN ":libro/a" AS 1 LEN=l
FIELD #1,l AS a$
GET l,1
ASS$=a$
CLOSE 1

ass&=SADD(ass$)
CALL ass&
END
```

Fig. 13.2: Metodo 1: Basic e Assembler separati

Il mio file code ha come nome A e si trova nella directory. Tramite la funzione LOF è possibile controllare quanto è lungo il file, e il risultato viene annotato nella variabile l.

Ora il file viene di nuovo aperto, ma questa volta come file Random. La lunghezza del record viene impostata affinché sia uguale a quella del file, per cui è sufficiente un “get” per poter leggere tale file (e ciò è importante). A questo punto il code si trova nella stringa ass$. Con SADD è possibile determinare ora l'indirizzo della stringa ed attribuirlo alla variabile ass&.

Fare attenzione all'operatore &! Con esso si produce forzatamente il tipo lungo integrale, cosa che è indispensabile per un indirizzo. A questo punto si può chiamare il programma macchina semplicemente con

```
        CALL Variabile Indirizzo
```

Come già detto, questo metodo ha lo svantaggio che è sempre necessario avere sul disco ambedue i moduli, il code e il programma in BASIC, cosa che spesso un utente (in caso di copiatura) purtroppo dimentica.
Un aiuto viene offerto dalla soluzione riportata in Figura 13.3.

```
DIM a%(64)

FOR i=1 TO 64
        READ a$:a%(i)=VAL("&h"+a$)
NEXT

ass&=VARPTR(a%(1))
CALL ass&
END

DATA 48E7,FEFE,43FA,0058,7000,2C79,0000,0004
DATA 4EAE,FDD8,4A80,6700,0044,2C40,41FA,004C
DATA 2208,243C,0000,03ED,4EAE,FFE2,2A00,4A80
DATA 6700,002A,2205,41FA,0050,2408,263C,0000
DATA 0001,4EAE,FFD6,2205,4EAE,FFDC,224E,2C79
DATA 0000,0004,4EAE,FE62,4CDF,7F7F,4E75,646F
DATA 732E,6C69,6272,6172,7900,434F,4E3A,3430
DATA 2F31,3030,2F35,3830,2F38,302F,6869,7420
DATA 616E,7920,6B65,7900
```

Fig. 13.3: Metodo 2: Basic e Assembler in un File

Il programma macchina si trova come stringa esadecimale nelle righe DATA. Con la semplice sequenza di istruzioni del Loop FOR, esso viene scritto in una matrice. Questa matrice deve essere del tipo integrale, diversamente il BASIC trasformerebbe le cifre in un formato con decimale fluttuante,dopodiché del nostro codice non resterebbe nulla.

La prima parola del codice si trova ora in a%(1). Con VARPTR è possibile determinarne l'indirizzo. Il resto è già noto. Ma come fa il codice ad andare nelle righe DATA?
La piccola utility, presentata in Figura 13.4, risolve questo problema, naturalmente in assembler, ed è per questo che ce ne occupiamo (in BASIC sarebbe più semplice, ma più noioso).

```
        opt     l-                      ;Solo con Assembler DevPac

* ABC = Assembler Basic Converter
* Object-File -> Righe di Basic-DATA
* (c) 1987 Peter Wollchlaeger
* --------------------------------

_SysBase          equ     4             ;Base di Exec
_LVOOpenLibrary   equ     -552          ;Apertura Library
_LVOCloseLibrary  equ     -414          ;Chiusura Library

_LVOOutput        equ     -60           ;Prelevamento dello StdOut-Handle
_LVOWrite         equ     -48           ;Scrittura (su File)
_LVORead          equ     -42           ;Lettura
_LVOOpen          equ     -30           ;Apertura File
_LVOClose         equ     -36           ;Chiusura
MODE_OLDFILE      equ     1005          ;File esistente
MODE_NEWFILE      equ     1006          ;File nuovo/riscrivere

* Alcune semplici Macro per il File-Handling
* --------------------------------------

OPEN    macro
        move.l  \1,d1                   ;Nome
        move.l  \2,d2                   ;Modo
        jsr     _LVOOpen(a6)
        endm

READ    macro
        move.l  \1,d1                   ;Handle
        move.l  \2,d2                   ;Indirizzo Buffer
        move.l  \3,d3                   ;Max.
        jsr     _LVORead(a6)
        endm

WRITE   macro
        move.l  \1,d1                   ;Handle
        move.l  \2,d2                   ;Indirizzo Buffer
        move.l  \3,d3                   ;Numero
        jsr     _LVOWrite(a6)
        endm

CLOSE   macro
        move.l  \1,d1           ;Handle
        jsr     _LVOClose(a6)
        endm
* --------------------------------------

        movem.l  a0/d0,-(sp)            ;Parametri riga di comando
```

```
* Apertura della DOS-Lib:

_main   move.l  #dosname,a1             ;Nome della DOS-Lib
        moveq   #0,d0                   ;Version indifferente
        move.l  _SysBase,a6             ;Base di  Exec
        jsr     _LVOOpenLibrary(a6)     ;Apertura di DOS-Lib
        tst.l   d0                      ;Errore?
        beq     fini                    ;se Errore, fine
        move.l  d0,a6                   ;Annotare il puntatore


* Riga di comando del Parser
* --------------------
        movem.l (sp)+,a0/d0             ;Prelevamento Parms
        move.b  #' ',-1(a0,d0.l)        ;terminare con spazio vuoto
                                        ; perche' il mio parser lo vuole
        lea     quelle,a1               ;Nome Buffer (File sorgente)
        bsr     parse                   ;Prelevamento
        lea     ziel,a1                 ;Nome file destinazione
        bsr     parse                   ;come sopra

* Apertura di ambedue i file
* -------------------
        OPEN    #quelle,#MODE_OLDFILE
        move.l  d0,d4                   ;Handle
        tst.l   d0                      ;qualcosa non ha funzionato?
        bne     weiter                  ;se no
        jsr     _LVOOutput(a6)          ;Prelev. dell Handle di Output
        WRITE   d0,#msg1,#len1          ;e segnalazione dell'errore
        bra     cl_lib                  ;quindi fine

weiter  OPEN    #ziel,#MODE_NEWFILE     ;come sopra, ma file destinazione
        move.l  d0,d5
        tst.l   d0
        bne     start
        jsr     _LVOOutput(a6)          ;Prelevamento Handle di Output
        WRITE   d0,#msg2,#len2
        bra     cl_q

* Qui cominciamo
* -------------------
start   READ    d4,#buffer,#16          ;Saltare l'Header

loop    READ    d4,#buffer,#16          ;Lettura di 16 Byte o meno
        tst.l   d0                      ;Fine del  File?
        beq     fertig                  ;se si

        asr     #1,d0                   ;Leggere 2 Byte
        subq    #1,d0                   ;- 1 a causa del dbra-loop
        move    d0,d6                   ;Annotare il numero delle parole

        WRITE   d5,#data,#6             ;write LF + 'DATA '

        lea     buffer,a4               ;Puntatore a sorgente
```

```
conv     move     (a4)+,d2                  ;una parola
         lea      hbuf,a0                   ;dovra' andare
         bsr      hex                       ;come stringa esadecimale
         WRITE    d5,#hbuf,#4               ;nel file di destinazione
         cmp      #0,d6                     ;Ultima parola nella riga?
         beq      no                        ;se no
         WRITE    d5,#komma,#1              ;diversamente una virgola dietro
no       dbra     d6,conv                   ;finche' la riga non e' finita
         bra      loop                      ;fino a  EOF
fertig   WRITE    d5,#LF,#1                 ;LF al termine


cl_z     CLOSE    d5                        ;Chiudi File di destinazione
cl_q     CLOSE    d4                        ;e il file sorgente

cl_lib   move.l   a6,a1                     ;Base di DOS-Lib
         move.l   _SysBase,a6               ;Base di Exec
         jsr      _LVOCloseLibrary(a6)      ;Funzione "Chiusura"

fini     rts                                ;Ritorno al CLI

* Parser semplice; riconosce solo spazi bianchi come delimitatori
* -----------------------------------------------------------------
parse    cmp.b    #' ',(a0)+                ;Spazi bianchi anteriori
         beq      parse                     ;da saltare
         subq.l   #1,a0                     ;eravamo troppo avandi di 1
p1       move.b   (a0)+,(a1)+               ;ora copia
         cmp.b    #' ',(a0)                 ;fino a spazio bianco successivo
         bne      p1
         clr.b    (a1)                      ;Termina con 0-Byte
         rts

* Conversione di  d2.w in Stringa ASCII da (a0)
* ----------------------------------------
hex
         moveq    #3,d1                     ;per 4 Nibble
next     rol      #4,d2                     ;Prelevamento di 1 Nibble
         move     d2,d3                     ;salvataggio in d3
         and.b    #$0f,d3                   ;mascheratura
         add.b   #48,d3                     ;trasformazione in ASCII
         cmp.b    #58,d3                    ;e' >9 ?
         bcs      out                       ;se no
         addq.b   #7,d3                     ;altrimenti deve essere A-F
out      move.b   d3,(a0)+                  ;Memorizzazione di 1 carattere
         dbra     d1,next                   ;next nibble
         rts

* -----------------------------------------------------
* Campo dati:

dosname  dc.b     'dos.library',0
         cnop     0,2
```

```
data     dc.b     10,'DATA '
         cnop     0,2
komma             dc.b   ','
LF       dc.b     10
msg1     dc.b     'File sorgente non trovato',10
len1     equ      *-msg1
         cnop     0,2
msg2     dc.b     'Non posso aprire il File di destinazione',10
len2     equ      *-msg2
         cnop     0,2

quelle   ds.b     40
ziel     ds.b     40
buffer   ds.b     16
hbuf     ds.b     4

         end
```

Fig. 13.4: Programma in Assembler che genera righe in Basic

Il programma chiamato ABC viene chiamato nel CLI con la sintassi

```
ABC Object_file Basic_file
```

L'Object_file è il programma macchina, esattamente come viene prodotto dall'assembler. Il Basic_file è quindi testo ASCII puro, nel quale ciascuna riga comincia con la parola DATA. Nelle righe DATA si trova il programma macchina sotto forma di stringhe. Ogni stringa è una parola in notazione esadecimale. La scrittura esadecimale ha il vantaggio di rendere più leggibile un programma.

Siccome abbiamo stabilito che in ogni riga ci devono essere otto parole, otteniamo il seguente schema per il programma:

1. Smembramento delle righe di comando nei nomi di file Sorgente e Destinazione
2. Apertura dei file
3. Scrittura di un Linefeed più la parola DATA più uno spazio bianco
4. Lettura di 16 byte
5. Trasformazione di 16 byte in otto stringhe esa
6. Scrittura di otto stringhe, suddivise da virgole
7. Ripetizione a partire dal punto 3 fine a EOF (sorgente)
8. Chiusura dei file

A questo punto incontriamo una piccola difficoltà: alla fine non restano 16 byte, per cui vale: leggi 16 byte o quel che resta, trasformane la meta in parole esadecimali.

Naturalmente si dovrà anche controllare se al momento dell'apertura del file qualcosa non ha funzionato e se per caso si è avuta una segnalazione di errore.

E con ciò avemmo descritto il compito, passiamo ora alla soluzione. Dopo l'inizio del programma, l'indirizzo della riga di comando (tutto il testo che segue il nome di programma) si trova nel registro A0 e la sua lunghezza in D0. E' normale, per prima cosa, salvare tali parametri, quindi verificare se si è in grado di aprire le loro libraries e quindi interpretare le righe di comando. Naturalmente ci si può risparmiare i due "movem", scrivendo l'apertura del DOS Lib dopo il "parser".

Come abbiamo già visto il Parser copia ambedue i nomi di file dalle righe di comando in due buffer e li conclude con un byte di 0 (è il DOS che lo vuole). A questo punto proviamo ad aprire i due file. In caso di errore, ci procureremo tramite "_LVOOutput" gli Handle dell'output standard (in questo caso la finestra CLI) e con la stessa Macro WRITE emetteremo la segnalazione di errore.

Questo è il motivo per il quale è possibile chiamare il programma solo dal CLI. Anche "RUN ABC..." non è permesso. Se lo si vuole a tutti i costi, si dovrà aprire una finestra come segue:

```
move.l  #window,d1              ;Indirizzo
move.l  #MODE_OLDfile,d2        ;Stato = c'è
jsr     _LVOOpen(a6)            ;Ora apertura
```

Con la nostra Macro la questione diventa ancora più semplice, cioè

```
OPEN    #window,#MODE_OLDfile
```

A questo punto, l'Handle si trova in D0 e può venire utilizzato sia per l'Input (Read) che per l'Output (Write). Nel campo dati manca ancora

```
window  dc.b    'CON:40/100/580/80/Titolo finestra',0
```

I quattro numeri descrivono l'angolo superiore sinistro della finestra, la sua larghezza ed altezza. Dopo ciò segue il titolo. Dopo la chiamata sarà necessario salvare I'Handle (D0). Quindi non dimentichiamo di richiudere la finestra. A questo punto qualcuno mi presenti un 68000 nel quale sia cosi facile aprire delle window (tramite software di sistema)! Tale finestra non ha la potenza di una window di Intuition, ma è molto comoda.

Dalla riga "Qui cominciamo" vengono prima di tutto letti 16 byte alla cieca, quindi inizia il loop già descritto di read/write. Infine i file vengono chiusi, dopodiché viene chiusa anche la DOS Library.

I 16 byte che abbiamo precedentemente ignorato sono il cosi detto file-Header. In essi troviamo una parola lunga con il contenuto $000003E7. Ciò significa che qui comincia un'unità di programma. Quindi segue una parola lunga con contenuto 0, sempre che non si sia dato nessun nome al modulo (è non è necessario). L'ultima parola lunga potrebbe essere il motivo per il quale si vogliono modificare le righe e si vogliono tralasciare solo 12 byte. In effetti in essa si trova la lunghezza del modulo in parola lunga meno uno.

Per quanto concerne i sottoprogrammi, vediamo che il Parser cerca un testo a partire dall'indirizzo in A0, che deve sempre cominciare e terminare con uno spazio bianco. L'ultimo spazio bianco è stato forzatamente determinato alla fine della riga di comando con

```
        move.b   #’ ‘,-1(a0,d0.l)
```

che d’altra parte rappresenta un bell’esempio per la potenza dei tipi di indirizzamento del 68000. Il testo isolato dal Parser viene copiato nel buffer a partire da A1. Dal momento che si tratta di un nome di file, i testi verranno terminati con un byte di zero.

La routine “esa” converte una cifra binaria in una stringa, che ne rappresenta il valore in esadecimale. Se si modifica il contatore di loop da 3 a 7 ed il “rol” in “rol.l” sarà anche possibile trasformare delle parole lunghe, come già spiegato al Capitolo 5

13.4 Attribuzione dei parametri

Fin qui la questione è stata abbastanza semplice, ma poco realistica. Normalmente si devono trasferire dei dati alle routine di assembler e da esse se ne vogliono prelevare i risultati. Ci chiederemo quindi come trasmettere parametri in ambedue le direzioni.

Risposta 1: E' necessario attribuire anche le variabili di risultato. Il programma assembler dovrà in seguito modificarle. Più esattamente: si forniscono gli indirizzi delle variabili BASIC e la routine macchina scriverà a partire da esse il risultato.

Risposta 2: Il trasferimento di parametri funziona tramite lo Stack.

Spieghiamolo meglio con un esempio. Ipotizziamo di volere che la routine assembler sostituisca con una x l'ultimo carattere di una stringa. Per fare ciò dovremo dare due parametri, cioè l'indirizzo della stringa e la sua lunghezza. In BASIC sarebbe come illustrato in Figura 13.5.

```
DIM a%(13),r%(3)
FOR i=1 TO 13
        READ a$:A%(i)=VAL("&h"+a$)
NEXT

a$="Ciao"
Indirizzo&=SADD(A$)
Lunghezza&=LEN(a$)

ass&=VARPTR(a%(1))
CALL ass&(Indirizzo&,Lunghezza&)
PRINT a$

END

DATA 48E7,8080,206F,000C,202F,0010,5380,11BC
DATA 0078,0800,4CDF,0101,4E75
```

Fig. 13.5: La stringa in Basic deve venire modificata

Come al solito, la routine viene caricata dalle righe Data, il suo indirizzo si trova quindi in ass&. L'indirizzo e la lunghezza della stringa vengono determinati, quindi ha luogo la chiamata con

```
        CALL    ass&(Indirizzo&,Lunghezza&)
```

Osserviamo ora cosa è contenuto nelle righe Data; la Figura 13.6 ce lo illustra.

```
        Opt      l+,p+
        movem.l  a0/d0,-(sp)                ;8 in piu' sullo Stack
                                            ;+ indirizzo di Return = 12

        move.l   12(sp),a0                  ;Indirizzo
        move.l   16(sp),d0                  ;Lunghezza
        subq.l   #1,d0
        move.b   #'x',0(a0,d0.l)

        movem.l  (sp)+,a0/d0
        rts
```

Fig. 13.6: Significato delle righe DATA in Fig. 13.5

Ora dobbiamo solo tenere presente che, tramite “movem”, sono pervenuti otto byte aggiuntivi sullo Stack (due registri) e che esiste ancora l'indirizzo di Return di 4 byte, per cui in totale abbiamo 12 byte.

Ora, a distanza di 4 e cominciando da 12, potremo accedere ai parametri in sequenza. Però facciamo attenzione, sarà semplice solo se tutti i parametri sono del tipo lungo. Dal momento che lo Stack cresce dall'indirizzo più alto all’indirizzo più basso, 12(sp) si trova più in basso di 16(sp). Ciò significa che il parametro di 12(sp) è l'ultimo nello Stack, che invece era il primo nella chiamata. In altre parole: i parametri vengono depositati in sequenza inversa.

Osserviamo quindi ancora un esempio. Come in Figura 13.7. una matrice o una parte di essa deve venire caricata molto velocemente con lo stesso valore in tutti gli elementi (o in quelli selezionati).

```
DIM a%(15),r%(100)
FOR i=1 TO 15
        READ a$:a%(i)=VAL("&h"+a$)
NEXT

ass&=0

Lunghezza&=5
Valore%=123
Indirizzo&=VARPTR(r%(3))

ass&=VARPTR(a%(1))
CALL ass&(Indirizzo&,Lunghezza&,Valore%)
FOR i=1 TO 10
        PRINT r%(i)
NEXT

END

DATA 48E7,C080,206F,0010,202F,0014C322F,001A
DATA 5358,30C1,51C8,FFFC,4CDF,0103,4E75
```

Fig. 13.7: Viene inizializzata una matrice

Dobbiamo ora fornire l'indirizzo dell'elemento a partire dal quale la matrice deve venire caricata (in questo caso r%(3)), la lunghezza (numero degli elementi) e naturalmente il valore. Il valore, essendo unico, è del tipo intero, cioè occupa solo 2 byte.

Però osserviamo che cosa mette il BASIC nello Stack. La chiamata è:

```
CALL ass&(Indirizzo&,Lunghezza&,Valore%)
```

Osserviamo la routine di assembler rappresentata in Figura 13.8

```
                  opt       l+,p+
                  movem.l   a0/d0-d1,-(sp)     ; 12 in piu' sullo Stack
                                               ;+ Indirizzo di Return =16
                  move.l    16(sp),a0          ;Indirizzo
                  move.l    20(sp),d0          ;Numero
                  move.w    26(sp).d1          ;Valore
                  subq.l    #1,d0              ;-1 a causa del dbra
loop              move.w    d1,(a0)+
                  dbra      d0,loop


                  movem.l   (sp)+,a0/d0-d1
                  rts
```

Fig. 13.8: Routine in Assembler per Fig. 13.7

E' chiaro che ora dobbiamo cominciare da 16. In conclusione 3 registri e l'indirizzo di Return si trovano sullo Stack con 4 byte ciascuno. Allora perché troviamo il valore a 26(sp) e non a 24(sp)? Evidentemente il BASIC non si fa influenzare dal simbolo di % (che dovrebbe significare Short Integer), e anche in quel caso mette 4 byte nello Stack. In effetti è possibile scrivere in BASIC anche "valore &" e prelevare il valore da 24(sp) come parola lunga. Questo fatto però ci creerebbe dei problemi in caso di numeri negativi, dal momento che il loro segno si trova per esempio nel bit 31, quindi lo perderemmo se prendessimo in una parola solo i bit da 0 a 15.

Se nei nostri esperimenti qualcosa non funziona, osserviamo il programma in BASIC. Io personalmente, verificando la routine, ho avuto un grosso insuccesso, in quanto invece di r% avevo scritto a%, cosa che ha avuto come conseguenza la riscrittura della routine in assembler con "valore" quando essa si trovava ancora a meta dell'esecuzione.

Inoltre si dovrà fare attenzione a non inserire nessuna nuova variabile nel BASIC se si è determinato un indirizzo in precedenza con VARPTR o SADD. La nuova variabile può spostare quella precedente, quindi il suo indirizzo può non coincidere.

Il sistema più semplice è quello di attribuire dapprima un valore a tutte le variabili. Di conseguenza anche “ass&=0”, rappresentato in Figura 13.7, ha un suo senso.

13.5 Chiamata in Basic di comandi CLI

Per concludere analizziamo una routine molto utile, che mette a disposizione il comando Shell dell'MS BASIC (IBM PC) anche per il BASIC per Amiga. Sappiamo già come utilizzare il comando Execute del DOS, nonché come fornire una stringa. Se si combina tutto ciò, se ne ottiene il programma di cui in Figura 13.9.

```
        opt     l+,p+    ;linkabile e position indipendent

* shell.s  Routine per la Chiamata dei comandi CLI in Basic
* (c) 1987 Peter Wollschlaeger

_SysBase         equ      4                 ;Basase di  Exec
_LVOOpenLibrary  equ      -552              ;Apertura Library
_LVOCloseLibrary equ      -414              ;Chiusura Library
_LVORead         equ      -42               ;Lettura File
_LVOOpen         equ      -30               ;Open File
_LVOClose        equ      -36
_LVOExecute      equ      -222              ;Execute CLI-Cmd

RELO    macro
        lea     \1(pc),a0                   ;Pgm deve diventare
        move.l  a0,\2                       ;position indipendent
        endm

        movem.l  d0-d6/a0-a6,-(sp)          ;14*4=56 Bytes
                                            ;+ Return-Adr. = 60
;
_main   lea     dosname(pc),a1              ;Nome della DOS-Lib
        moveq   #0,d0                       ;Versione indifferente
        move.l  _SysBase,a6                 ;Base di Exec
        jsr     _LVOOpenLibrary(a6)         ;Apertura della DOS-Lib
        tst.l   d0                          ;Errore?
        beq     fini                        ;se si, fine
        move.l  d0,a6                       ;annotare il puntatore

        RELO    name,d1                     ;Nome della DOS-Lib
        move.l  #1005,d2                    ;Stato = c'e'
        jsr     _LVOOpen(a6)                ;or apertura
        move.l  d0,d5                       ;annotare Handle
        tst.l   d0                          ;Errore?
        beq     fini                        ;se si, interruzione
```

```
        move.l   60(sp),a0                  ;indirizzo della Basic-String
        move.l   64(sp),d0                  ;Lunghezza

        subq.l   #1,d0                      ;-1 a causa del dbra
        lea      buffer(pc),a1              ;qui
loop    move.b   (a0)+,(a1)+                ;copiare
        dbra     d0,loop
        move.b   #0,(a1)                    ;terminare con 0

        RELO     buffer,d1                  ;Indirizzo del comando  CLI
        clr.l    d2                         ;nessun Input
        move.l   d5,d3                      ;Handle della Window
        jsr      _LVOExecute(a6)            ;chiamata del CLI

        move.l   d5,d1                      ;lettura da CON
        RELO     buffer,d2                  ;in questo buffer
        move.l    #1,d3                     ;1 carattere
        jsr      _LVORead(a6)               ;Chiamata di lettura

        move.l   d5,d1                      ;chiusura di CON
        jsr      _LVOClose(a6)

        move.l   a6,a1                      ;Base di DOS-Lib
        move.l   _SysBase,a6                ;Base di Exec
        jsr      _LVOCloseLibrary(a6)       ;Funzione "Chiusura"
fini    movem.l  (sp)+,d0-d6/a0-a6


        rts                                 ;Ritorno al CLI

dosname dc.b     'dos.library',0
        cnop     0,2
name    dc.b     'CON:10/20/600/160/Con un tasto a piacere -> Basic',0
        cnop     0,2
buffer  dc.b     'ABCDE'                    ;per poterlo trovare
        end
```

Fig. 13.9: Emulazione del comando Shell

In pratica non facciamo nient'altro che copiare in un buffer la stringa fornita dal BASIC e chiuderla con un byte di zero. Quindi possiamo chiamare il _LVOExecute. L'Output ha luogo in una propria finestra. Dopo di ché si attenderà la battitura di un tasto (lettura di un carattere), e si ritornerà al BASIC. Il listato è mostrato in Figura 13.9.

```
* Comando Shell in Basic per Amiga
* (c) 1987 Peter Wollshlaeger

DIM assem%(140): InitShell

shell "dir jh0:" 'Demo

END

SUB InitShell STATIC
        SHARED assem%()
        FOR i=1 TO 96
                READ a$:assem%(i)=VAL("&h"+a$)
        NEXT
END SUB

SUB shell(a$) STATIC
        SHARED assem%()
        Indirizzo&=SADD(A$):Lunghezza&=LEN(a$)
        ass&=VARPTR(a%(1))
        CALL ass&(Indirizzo&,Lunghezza&)
END SUB

DATA 48E7,FEFE,43FA,007E,7000,2C79,0000,0004
DATA 4EAE,FDD8,4A80,6700,0066,2C40,41FA,0072
DATA 2280,243C,0000,03ED,4EAE,FFE2,2A00,4A80
DATA 6700,004C,206F,003C,202F,0040,5380,43FA
DATA 0080,12D8,51C8,FFFC,12BC,0000,41FA,0072
DATA 2208,4282,2605,4EAE,FF22,2205,41FA,0062
DATA 2408,263C,0000,0001,4EAE,FFD6,2205,4EAE
DATA FFDC,224E,2C79,0000,0004,4EAE,FE62,4CDF
DATA 7F7F,4E75,646F,732E,6C69,6272,6l72,7900
DATA 434F,4E3A,3130,2F32,302F,3630,302F,3136
DATA 302F,4D69,7420,6265,6C69,6562,6967,6572
DATA 2054,6173,7465,202D,3E20,4261,7369,6300
```

Fig. 13.10: Parte in BASIC di figura 13.9

Con qualche trucco abbiamo risolto la questione del buffer. Dal momento che esso do-vrebbe essere esattamente di 80 caratteri, ma che un “ds.b 80” avrebbe allungato le righe Data di 40 parole con “0000”, non abbiamo nessun altro problema nel BASIC per quello che riguarda il buffer. Nel listato in assembler avevamo marcato |'inizio del buffer con “dc.b ’ABCDE'”. Lo abbiamo fatto solo per poter trovare facilmente la posizione nelle righe Data prodotte originariamente da ABC. A partire da tale parola (AB) abbiamo quindi cancellato tutti i dati in BASIC. La matrice, invece, con 140, è stata dimensionata in maniera maggiore di quanto il codice (96 parole) lo richieda. I| resto serve come buffer, e questo è il sistema per risolvere tale problema.

A questo punto resterebbe ancora una questione: in questa soluzione, la routine in assembler attende sempre la pressione di un tasto prima di tornare al BASIC. Ciò ha naturalmente senso se si vuole anche vedere il risultato, per esempio di un comando Dir, ma disturba se si utilizza “Shell” in un programma in funzione, che serva per esempio per copiare dei file. La mia proposta sarebbe: sostituiamo semplicemente il “jsr _LVORead(a6)” con “NOP NOP”. NOP significa No Operation (non fare nulla) ed ha il codice 4E71.

Nel listato ambedue le parole Dato da riscrivere con dei NOP sono sottolineate. Questa mia proposta può venire risolta nel modo più semplice tramite il BASIC. Scriviamo due sottoprogrammi Shell e ShellWait. Il primo scriverà dapprima (numericamente) $4EAE, $FFD6 negli elementi giusti della matrice, il secondo scriverà due volte $4E71.

CAPITOLO 14

Exec e DOS in dettaglio

Processi e Task
Exec, il Capo
DOS, Intuition, Library e Device
Il DOS in pratica

14.1 Processi e Task

Finora abbiamo sempre solo parlato di task. In contrapposizione ad essi, in Amiga abbiamo dei processi, cosa che forse può creare un po' di confusione, in quanto spesso i due vocaboli assumono lo stesso significato. Proviamo a spiegare meglio: nel presente contesto un processo è un "sovra-task". La conseguenza di ciò è che sotto un processo possono girare (quasi) contemporaneamente diversi task. Un task può chiamare solo un altro task. il quale girerà al posto di quello che l'ha chiamato.

Se osserviamo le strutture dei dati illustrate in appendice, stabiliremo immediatamente che un blocco di controllo del task non è altro che un sottogruppo (un estratto) di un blocco di controllo di processo. Accederemo molto raramente a queste strutture; ma quando lo faremo dovremo sapere a quale accedere. Nel Capitolo 9 abbiamo osservato l'UCP (in italiano Unita di Controllo di Processo) dove si trova il nostro task. L'accesso diretto, inteso come modifica di parametri di queste strutture, è sempre rischioso (infatti si deve tenere conto di moltissime condizioni di contorno). Ecco perché esistono delle funzioni anche per tutte le applicazioni importanti, come per esempio la modifica della priorità di un task.

14.2 Exec, il Capo

Exec è l'abbreviazione di Executive,che in americano significa Capo (dirigente). Exec è naturalmente un processo che gira sempre (quando l'Amiga è acceso).

Multitasking

E' l'Exec stesso a costituire il sistema Multitasking dell'Amiga. Tale sistema ha il compito di distribuire le risorse di un sistema sui diversi task. Le risorse sono la CPU (il 68000), la memoria principale e le apparecchiature periferiche. Le risorse non divisibili come la CPU vengono attribuite secondo un procedimento di priorità. Verrà elaborato per primo il task con la priorità più alta, quindi quello immediatamente successivo. Se diversi task hanno la stessa priorità, essi verranno elaborati in intervalli secondo un procedimento di "time-sharing". Tramite Interrupt di Hardware, Exec ha sempre il controllo della situazione, anche quando un task non vorrebbe terminare la propria attività.

Un task ha tre stati, cioè funzionante,non funzionante e in attesa. Se si tiene conto degli altri task, un task dovrebbe venire impostato allo stato di attesa piuttosto spesso (attesa di input). Come ciò accade è stato spiegato nel Capitolo 12 (Event). Se non si chiama _LVOWait, ma si interroga continuamente _LVOGetMsg (spesso inutilmente) sulla presenza di una notizia, si sprecherà del tempo inutilmente, che verrà a mancare

agli altri task. E' importante sottolineare questo fatto, in quanto io personalmente ho visto moltissimi listati che presentano tale stile di programmazione poco adeguato.

Gestione della Memoria

La risorsa RAM viene attribuita permanentemente a ciascun task, per quanto concerne la memorizzazione del codice programma e dei dati. Diventa invece pericoloso il fatto che un task necessiti di memoria (allochi) in maniera dinamica (durante il funzionamento del programma). La sua necessità di memoria non potrà venire adempiuta se altri task l'hanno già occupata tutta. Di conseguenza sarà opportuno occupare (cercare di occupare) all'inizio del programma la quantità di memoria di cui si avrà bisogno, e non lasciare intraprendere all'utente diversi input.

Lo stesso vale per le library. E' necessario aprire una library con _LVOOpen library.
A questo punto Exec controllerà se questa Lib è già stata caricata (oppure se si trova nella ROM o nella Kick-start-RAM). Se non lo è, Exec cerca di caricare la Lib dal disco. Se manca la memoria per essa, Exec fornirà uno zero come Lib Basis. Di conseguenza il programma dovrà aprire all'inizio anche tutte le Library, e non solo quando ce n'è bisogno, come si vede talvolta. Infatti, a quel punto, potrebbe essere troppo tardi. In ogni caso sarà sempre necessario chiudere immediatamente una Lib quando non serve più. Se noi infatti siamo gli unici o gli ultimi utenti di questa Lib, Exec libererà di nuovo lo spazio di memoria che questa occupava.

14.3 DOS, Workbench, Intuition, Library e Device

Il sistema operativo dell'Amiga è suddiviso in diversi moduli, correlati gerarchicamente l'uno all'altro. La figura 14.1 tenta di spiegare questa correlazione.

Per noi prima di tutto è importante che la nostra applicazione trovi sempre la via per l'Hardware. Di conseguenza un programma di Workbench dovrà dapprima aprire Intuition, al fine di poter per esempio produrre una window. Se in questa window vogliamo disegnare, avremo bisogno per lo meno di Graphics, quindi ne dovremo aprire la Lib.

Anche l'AmigaDOS è un processo, sotto il quale girano diversi task. Per esempio il Disk-Validator, che gira sempre, è un task che verifica continuamente tramite letture di prova se un dischetto si trova inserito in tutte le unità a disco.

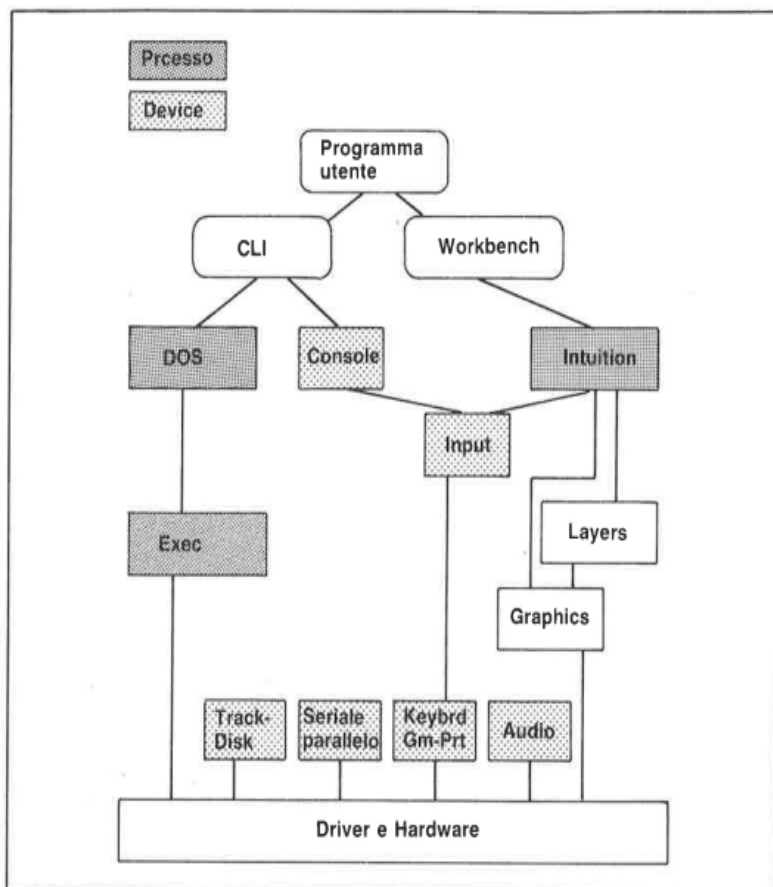



Fig. 14.1: Interconnessione dei moduli Software dell'Amiga

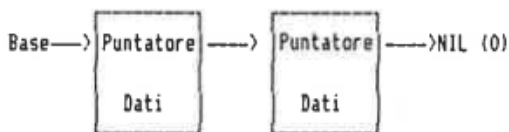
Il Workbench ed il CLI sono livelli utente con lo stesso grado di abilitazione, cioè praticamente dei programmi che interpretano gli input dell'utente e che li eseguono all'interno delle loro possibilità. La partenza dell'Amiga con l'uno o l'altro dei due dipende principalmente dal testo che si trova nella "Startup sequence", con la quale l'Amiga è superiore a molti altri elaboratori, nei quali questa alternativa di start può venire ottenuta solo tramite una modifica diretta dei settori di Boot del dischetto di partenza, cosa che presuppone delle conoscenze molto approfondite del sistema.

Non è più necessario che vi presenti Intuition, occupiamoci solo dell'ordinamento.

Device

I Device (gestori di apparecchiature) costituiscono l'interfaccia verso l'hardware. Quindi anche questi gestori sono dei task e superano di molto i gestori tradizionali. Un gestore normale di periferica mette solo a disposizione l'interfaccia, cioè traduce per esempio i comandi DOS in una serie di bit richiesta da un determinato hardware. Quindi nel DOS è sempre necessario fare in modo che venga effettuata la trasmissione carattere per carattere, per esempio alla stampante. Con l'Amiga è diverso: infatti forniamo un compito al task (device) senza più occuparcene.

Strutture e Liste

Come avrete già capito, senza strutture non è possibile fare nulla con l'Amiga (vedi anche Capitolo 11). Quasi tutte le strutture vengono gestite sotto forma di liste, come rappresentato anche nella figura seguente.

Un elemento di una lista di questo genere viene chiamato nodo. Un nodo è sempre composto da (almeno) un puntatore, che punta al nodo successivo, e da dati. L'ultimo nodo punta a NIL (nulla), per il quale in assembler si utilizza un semplice zero. Conosciamo già queste strutture, per esempio dai menu.

Nello stesso modo Exec gestisce anche i task. Ogni task ha un TCB (Task Control Block = blocco di controllo del task) i cui dati si trovano in una lista di questo genere.
La differenza sta solo nel fatto che le liste Exec sono sempre puntate in maniera doppia, cioè hanno sempre un puntatore al predecessore. Con ciò è possibile realizzare più in fretta operazioni tipiche di gestione di task, come per esempio la messa in ordine dei task a seconda delle priorità oppure l'introduzione o eliminazione di nodi (task).
Il lettore interessato faccia attenzione al file Include "nodes.i" (presente nel Metacomco e in HiSoft), nel quale sono contenute delle Macro ottimali per la gestione delle liste.

14.4 DOS ed Exec in pratica

14.4.1 Directory

Per quanto concerne il DOS, dobbiamo occuparci di un aspetto che non è stato ancora citato, cioè del funzionamento dl "Lock" e "FIB".

L'accesso completo ad un file o ad una directory è possibile solo tramite FIB (File Info Block). Al fine di poter accedere a questo FIB, è necessario procurarsi dapprima un Lock (chiave), tramite la quale sia possibile aprire il FIB.

Osserviamo un attimo il FIB da un estratto dell'Appendice 4, nella quale sarà possibile trovare ulteriori informazioni

```
00      fib_DiskKey      ds.l     1
04      fib_DirEntryType ds.l     1                ;0=file, >0 = Dir.
08      fib_fileName     ds.b     108              ;Tuttavia  max. 30
74      fib_Protection   ds.l     1                ;Ved. equ sotto
78      fib_EntryType    ds.l     1
7C      fib_SIze         ds.l     1                ;Dimensione file
80      fib_NumBlocks    ds.l     1
84      fib_DateStamp    ds.b     ds_SIZEOF        ;Ultima modifica
90      fib_Comment      ds.b     116
        fib_SIZEOF       equ      $104
```

Fig. 14.2: Struttura del file Info Black

Vediamo quindi che in esso si trovano tutte le informazioni relative ad un file, come il nome, la dimensione o il tipo. La struttura di "date _stamp" (data creazione/modifica) si trova anche in Appendice 4.

Con un piccolo programma che elenchi una directory, approfondiamo meglio come lavorare con queste informazioni, come rappresentato in figura 14.3. E non ditemi che sarebbe sufficiente battere semplicemente DIR in CLI. Come faremo invece con un programma che deve controllare se una directory ed un file esistono?

```
        opt     l-              ;non linkare!

* DIR_2 Visualizza directory usando le funzioni DOS

_LVOLock        equ     -84
_LVOExamine     equ     -102
_LVOExNext      equ     -108
_LVOIoErr       equ     -132
ERROR_NO_MORE_ENTRIES   equ     232

        include OpenDos.i               ;vedi capitolo  4

        jsr     _LVOOutput(a6)          ;Ottieni l'handle di output
        move.l  d0,d4                   ;e ricorda

        move.l  #pfad,d1                ;Indirizzo percorso
        move.l  #-2,d2                  ;Accesso in lettura
        jsr     _LVOLock(a6)            ;ottenere Lock della directory
        tst.l   d0                      ;c'e¨ una directory?
        beq     fertig                  ;se no
        move.l  d0,d5                   ;altrimenti ricorda Lock

        move.l  d5,d1                   ;Lock
        move.l  #fib,d2                 ;Indirizzo FIB
        jsr     _LVOExamine(a6)         ;Ottieni il primo nome (disco)
        tst.l   d0                      ;Trovato?
        beq     fertig                  ;se no
        bsr     print                   ;altrimenti stampa

loop    move.l  d5,d1                   ;Lock
        move.l  #fib,d2                 ;FIB
        jsr     _LVOExNext(a6)          ;Cerca il prossimo file
        tst.l   d0                      ;Trovato?
        beq     fertig                  ;se no
        bsr     print                   ;altrimenti stampa
        bra     loop                    ;finche' non c'e' piu' nulla


fertig  jsr     _LVOIoErr(a6)           ;Ottieni il codice di errore
        cmpi.l  #ERROR_NO_MORE_ENTRIES,d0     ;questo errore?
        beq     f1                      ;se si

        move.l  d4,d1                   ;Output-Handle
        move.l  #err,d2                 ;Testo
        move.l  #err_len,d3             ;d3
        jsr     _LVOWrite(a6)           ;output nome

f1      move.l  a6,a1                   ;Chiusura DOS-Lib
        move.l  _SysBase,a6
        jsr     _LVOCloseLibrary(a6)
fini    rts
```

```
print   lea     fib+8,a0                ;Nome
        lea     buffer,a1               ;copia nel Buffer
        moveq   #1,d3                   ;Contatore lunghezza
p1      addq.l  #1,d3                   ;incremento
        move.b  (a0)+,(a1)+             ;Byte di zero copiato?
        bne     p1                      ;se no
        move.b  #10,(a1)                ;aggiungere LF
        move.l  d4,d1                   ;Output-Handle
        move.l  #buffer,d2              ;Nome Indirizzo
        jsr     _LVOWrite(a6)           ;output nome
        rts

* Campo dati

dosname dc.b    'dos.library',0
        cnop    0,2
buffer  ds.b    108
pfad    dc.b    'df0:',0
        cnop    0,2
fib     ds.b    $104                    ;vedere l'Appendice A4.2
err     dc.b    10,'Qualcosa non va!',10
err_len equ     *-err
```

Fig. 14.3: Accesso alla directory

E' possibile ottenere il Lock molto semplicemente nella maniera seguente:

```
        move.l  #pfad,d1                ;Indirizzo percorso
        move.l  #-2,d2                  ;Accesso in lettura
        jsr     _LVOLock(a6)            ;ottenere Lock della directory
```

Come al solito il Lock si trova in d0, ma salviamolo immediatamente in d5. A questo punto è possibile chiamare delle funzioni che si aspettano come parametro il Lock e l'indirizzo del FIB (che per il momento è solo un campo di memoria vuoto). Un esempio di ciò sarebbe (Lock in d5):

```
        move.l  d5,d1                   ;Lock
        move.l  #fib,d2                 ;Indirizzo FIB
        jsr     _LVOExamine(a6)         ;Ottieni il primo nome (disco)
```

_LVOExamine controlla se il percorso (in questo caso il nome di disco) esiste e lo scrive, in caso positivo, nel FIB. In esso il nome si trova a partire da FIB + 8 (vedi figura 14.2). Non sappiamo quanto è lungo il nome, ma solo che termina con un byte di zero. La routine “print” conta dapprima i caratteri fino al byte di zero, quindi chiama il già noto _LVOWrite.

Come in ogni DOS, a questo punto troviamo una funzione che cerca ulteriori immissioni, in questo caso essa si chiama _LVOExNext (Examine Next). Se non c'è altro, essa ritorna a zero, diversamente è possibile terminare il nome.

Proviamo era a far uscire anche la dimensione del file. Con

```
        move.l fib+$7C,d2
```

ne portiamo il valore in d2, dove c'è “Bindec” che lo aspetta. Tuttavia non dimentichiamo di andare a prendere LF in “print” e di farlo uscire dopo l'output dei numeri.
A questo punto, con “EntryTyp” è possibile determinare se, nel caso del nome, si tratta di una directory.

14.4.2 Chiamata di Comandi CLI

Se si vuole solo visualizzare la directory, il programma sarà molto più semplice. Infatti sarà possibile chiamare qualunque comando CLI, quindi anche DIR, a partire da un programma. Rimaniamo un attimo su DIR, la figura 14.4 ce ne mostra la soluzione

```
        opt     l-              ;non linkare!

* DIR_1 Visualizzazione della Directory per mezzo di Execute

_LVOExecute     equ     -222
        include OpenDos.i                ;vedi capitolo 4

        move.l  #string,d1               ;Indirizzo del comando CLI
        clr.l   d2                       ;Nessun Input
        clr.l   d3                       ;Output nella Window CLI
        jsr     _LVOExecute(a6)          ;chiamata di CLI

        move.l  a6,a1                    ;Chiusura di DOS-Lib
        move.l  _SysBase,a6
        jsr     _LVOCloseLibrary(a6)

fini    rts

* Campo dati

dosname dc.b    'dos.library',0
        cnop    0,2
string  dc.b    'dir',0
```

Fig. 14.4: Chiamata di comandi del CLI, (qui DIR) da un programma

La funzione _LVOExecute richiede come parametri prima di tutto l'indirizzo (in d1), a partire dal quale è posizionato il comando per esteso. In d2 e d3 devono esserci degli zeri oppure degli Handle di file, come li si ottiene da _LVOOpen. Se in d2 si trova l'Handle di un file di input, tale file verrà letto ed il suo contenuto verrà interpretato come sequenza di comandi.

Il primo comando a venire eseguito sarà naturalmente quello identificato con d1. Dal momento però: che in d1 non ci deve essere un comando, sarà possibile riempire di zeri anche d1. Ciò significa che a questo punto verrà eseguita solo la sequenza di comandi che si trova nel file identificato con d2. Vediamo ora cosa fa il comando Execute del CLI.

Se in d3 c'è uno zero, l'output ha luogo nella finestra CLI corrente, diversamente avrà luogo nel file il cui Handle è presente in d3.

14.4.3 Exec

Esiste una stretta correlazione tra DOS e Exec. Il primo aspetto di tale correlazione è che naturalmente anche il DOS viene controllato dall’Exec, un altro aspetto è che molti comandi DOS sono molto pigri, cioè inoltrano il lavoro ad Exec.

Ciò vale per tutto l'IO (Input, Output), sia che si faccia riferimento ai dischetti, alla stampante, ecc. In linea di massima l’IO funziona in modo tale che un Request-Block di IO viene caricato con parametri. Poi viene chiamato (requested) un device-task, che dovrà svolgere il lavoro, quindi Exec inoltra ulteriormente il lavoro. Di conseguenza, lo svolgimento è sempre il seguente:

1. Inizializzazione del Request-Block
2. Apertura del Device
3. Chiamata di DoIO oppure SendIO
4. Chiusura di Device

Con DoIO la Request viene inviata al Device Task, che attende che un evento sia presente (valore di funzione OK, risultato nel buffer). Con SendIO avviene dapprima la stessa cosa, ma non c'è attesa. Il programma continua a girare ed il task gira in parallelo ad esso. Nel presente testo non possiamo approfondire ulteriormente questo argomento, in quanto esso sarebbe interessante solo per casi specifici. Per i casi normali abbiamo a disposizione delle funzioni molto comode e delle routine già pronte.

APPENDICE

Appendice A1: Elenco dei comandi del 68000

I comandi vengono rappresentati in un formato compatto, come nell'esempio seguente:

ADD ea,Dn / ADD Dn,ea — Addizionare
B W L — S + D --> D

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.w	$.L	d(PC)	d(PC.Rn)	#
S:	*	*	WL	*	*	*	*	*	*	*	*	*
D:			*		*	*	*	*	*	*		

Nella prima riga si trova sempre il comando nelle forme sintattiche permesse, seguite da una breve spiegazione.

Dn	è un registro dati a piacere
An	è un registro indirizzi a piacere
Rn	è un indirizzo dati o registri a piacere
S	è l'operando di sorgente (Source)
D	è l'operando di destinazione (destination)
#K	è una costante
d	é una distanza di indirizzamento

Nella seconda riga si trovano le dimensioni permesse degli operandi (B, W, L). Sotto di loro sono elencati i tipi possibili di indirizzamento. Un “*” significa che tutte le dimensioni di operandi precedentemente elencate sono permesse anche per questo indirizzamento. Una o due lettere delimitano il tipo di indirizzamento per tali dimensioni di operandi.

La sigla cc, per esempio in DBcc, significa Condition Code. Il suo significato è illustrato nell'ultima pagina della presente appendice.

ABCD Dn,Dn / ABCD -(An),-(An) — Addizionare BCD
B — S + D + X→ D

ADD ea,Dn / ADD Dn,ea — Addizionare
B W L — S + D→ D

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
S:	*	*	WL	*	*	*	*	*	*	*	*	*
D:			*	*	*	*	*	*	*			

ADDA ea,An — Addizionare Indirizzo
W L — S + D → D

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
S:	*	WL	*	*	*	*	*	*	*	*	*	*

L’operando Parola viene ampliato con EXT.L

ADDI #K,ea — Addizionare Costante
B W L — #K + D → D

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
S:	*		*	*	*	*	*	*	*			

ADDQ #K,ea — Addiz. Costante Quick (#K < = 8)
B W L — #K + D → D

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
D:	*	WL	*	*	*	*	*	*	*			

ADDX Dn,Dn / ADDX -(An),-(An) — Addizionare con Flag X
S + D + X → D

AND ea,Dn / AND Dn,ea — AND logico
B W L — S AND D → D

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
S:	*			*	*	*	*	*	*	*	*	*
D:			*	*	*	*	*	*	*			

ANDI #K,ea — AND logico con costante
B W L — # K AND D → D

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
D:	*		*	*	*	*	*	*	*			

ANDI #K,CCR — Operazione di AND a CCR
B — #K AND CCR→ CCR

ANDI #K,SR — Operaz. di AND a SR !Privilegiata!
W — #K AND SR → SR

ASL Dn,Dn / ASL #K,Dn / ASL ea — Spostam. aritmetico verso sinistra
B W L — D spostato di n Bit → D

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
D:			*	*	*	*	*	*	*			

0 viene spostato, il Bit fatto uscire va nel flag C e X

ASR Dn,Dn / ASR #K,Dn / ASR ea — Spostam. aritmetico verso sinistra
B W L — D spostato di n. Bit → D

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
D:			*	*	*	*	*	*	*			

Il Bit MS si sposta, ma resta. Il Bit che viene fatto uscire va nel Flag C e X

Bcc Label — Dirama se cc (relativo al PC)
.B W — Ved. Tabella cc
.S — PC + d → PC

BCHG Dn,ea / BCHG #K,ea — Prova e modifica del Bit n

B L

Bit-Test → Z-Flag

Modifica del Bit

Se DN Source: n = 0..31, diversamente 0..7. Se la destinazione è in Ram, viene sempre letto un Byte ed n = n mod 8.

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
D:	L		B	B	B	B	B	B	B			

BCLR Dn,ea / BCLR #K,ea — Prova e cancellazione Bit n

B L

Bit-Test → Flag Z

Bit = 0

Se DN Source: n = 0..31, diversamente 0..7. Se la destinazione è in Ram, viene sempre letto un Byte ed n = n mod 8.

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
D:	L		B	B	B	B	B	B	B			

BRA Label — Dirama a Label (relativa a PC)

.B W

.S

PC + d → PC

BSET Dn,ea / BSET #K,ea — Prova e impostazione del Bit n

B L

Bit-Test → Flag Z

Bit = 1

Se DN Source: n = 0..31, diversamente 0..7. Se la destinazione è in Ram, viene sempre letto un Byte ed n = n mod 8.

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
D:	L		B	B	B	B	B	B	B			

BSR Label — Call Sub con Label (relativa al PC)

.B W

.S

PC→ -(SP); PC + d → PC

BTST Dn,ea / BSET #K,ea — Prova del Bit n

B L — Bit-Test → Flag Z

Se DN Source: n = 0..31, diversamente 0..7. Se la destinazione è in Ram, viene sempre letto un Byte ed n = n mod 8.

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
S:	L		B	B	B	B	B	B	B	B	B	B
D:	L		B	B	B	B	B	B	B	B	B	

CHK ea,Dn — Controllo del registro rispetto ai limiti

W — if Dn <0 or Dn >(ea) then trap

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
S:	W		W	W	W	W	W	W	W	W	W	W

CLR ea — Cancellazione operando

B W L — 0 → D

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
D:	*		*	*	*	*	*	*	*			

CMP ea,Dn — Confronto operandi

B W L — Flag come da D meno S

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
S:	*	W	*	*	*	*	*	*	*	*	*	*

CMPA ea,An — Confronto indirizzi

W L — Flag come D meno S

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
S:	*	*	*	*	*	*	*	*	*	*	*	*

Operando Parola precedentemente ampliato a Lungo

CMPI #K,ea — Confronto rispetto a costanti
B W L — Flags come da D meno S

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
D:	*		*	*	*	*	*	*	*			

CMPM (An) +, (An) + — Confronto posizioni di memoria
B W L — Flag come da D meno S

DBcc Dn,Label — Test di cc. Decremento di Dn.
Branch if cc – false
then Dn = D n - 1
if Dn <> - 1 then BRA Label
else << avanti >>

DIVS ea,Dn — Divisione Parole Signed
W — D/S → D

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
S:	*		*	*	*	*	*	*	*	*	*	*

Quoziente nella parola di valore più basso, il resto in quella a valore più alto

---

DIVU ea,Dn — Divisione Parole Unsigned
W — D/S → D

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
S:	*		*	*	*	*	*	*	*	*	*	*

Quoziente nella parola di valore più basso, il resto in quella a valore più alto

---

EOR Dn,ea — XOR logico
B W L — S xor D → D

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
D:	*		*	*	*	*	*	*	*			

EORI #K,ea — XOR logico con costante
B W L — S xor D → D

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
D:	*		*	*	*	*	*	*	*			

EORI #K,CCR — XOR costante con CCR
B — S xor CCR → CCR

EORI #K,SR — XOR costante con SR !Privileg.!
W — S xor CCR → CCR

EXG Rn,Rn — Scambio registri
L — Rn ↔ Rn

EXT Dn — Ampliamento di Dn con correzione di segno
W L

ILLEGAL — Cancella le eccezioni non permesse

JMP ea — Salto assoluto (lungo)
D → PC

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
D:			*			*	*	*	*	*	*	

JSR ea — Chiamata assoluta di UP
PC → (SP); D → PC

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
D:			*			*	*	*	*	*	*	

LEA ea,An — Caricamento indirizzo effettivo

L — D → An

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
D:			*			*	*	*	*	*	*	

LINK An,#d — Indirizzamento Stack locale

An → -(SP); SP→ An; SP + d → SP

LINK e UNLK vengono utilizzati al fine di applicare una »linked list« di variabili locali per delle chiamate di UP inscatolate

---

LSL Dn,Dn / LSL #K,Dn / LSL ea — Spostamento logico a sinistra

B W L — D spostato di n Bit → D

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
D:			*	*	*	*	*	*	*			

0 viene spostato, il Bit fatto uscire va nei Flag C e X

LSR Dn,Dn / LSR #K,Dn / LSR ea — Spostamento logico a destra

B W L — D spostato di n Bit → D

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
D:			*	*	*	*	*	*	*			

0 viene spostato, il Bit fatto uscire va nei Flag C e X

MOVE ea,ea — Copiatura di dati

B W L — D → S

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
S:	*	WL	*	*	*	*	*	*	*	*	*	*
D:	*		*	*	*	*	*	*	*			

MOVE CCR,ea — Presa di CCR

W — CCR → ea

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
D:	*		*	*	*	*	*	*	*			

MOVE ea,CCR — Caricamento di CCR

W — ea → CCR

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
S:	*		*	*	*	*	*	*	*	*	*	*

MOVE ea,SR — Caricamento di SR !Privileg.!

W — ea → SR

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
S:	*		*	*	*	*	*	*	*	*	*	*

MOVE SR,ea — Presa di SR !Privileg.!

W — SR → ea

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
S:	*		*	*	*	*	*	*	*	*	*	*

MOVE USP,An — Presa di USP !Privileg.!

L — USP → An

MOVEA ea,An — Copiatura di indirizzo

L — ea → An

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
S:	*	*	*	*	*	*	*	*	*	*	*	*

MOVEM R_Lista,ea / MOVEM ea,R_Lista — Copiatura lista registri
W L

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
1:			*		*		*	*	*			
2:			*	*		*	*	*	*	*	*	

1= Registro → Memoria, es.: movem d1-d3/a1-a4,-(a7)
2= Memoria → Registro es.: movem (a7) + ,d1-d3/a1-a4

MOVEP Dn,d(An) / MOVEP d(An),Dn — Dati da/a periferiche
W L
I dati vengono trasferiti Byte per Byte

MOVEQ #K,Dn — Trasferimento »Quick«
L — #K(8 Bit) → Dn

MULS ea,Dn — Moltiplicazione con segno
W — S*D → D

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
S:	*		*	*	*	*	*	*	*	*	*	*

MULU ea,Dn — Moltiplicazione senza segno
W — S*D → D

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
S:	*		*	*	*	*	*	*	*	*	*	*

NBCD ea — Negiere BCD-Zahl
B — 0-D-X → D

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
S:	*		*	*	*	*	*	*	*	*	*	*

NEG ea — Negazione operando

B W L — 0-D → D

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
D:	*		*	*	*	*	*	*	*			

NEGX ea — Negazione operando con Flag X

B W L — 0-D-X → D

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
D:	*		*	*	*	*	*	*	*			

NOP — No Operation

non fa nulla (durata di 4 cicli di clock)

NOT ea — No logico

B W L — -D → D (Complemento di uno)

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
D:	*		*	*	*	*	*	*	*			

OR ea,Dn / OR Dn,ea — O logico

B W L — S or D → D

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
S:	*		*	*	*	*	*	*	*	*	*	*
D:			*	*	*	*	*	*	*			

ORI #K,ea — O logico con costante

B W L — # K or D → D

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
D:	*		*	*	*	*	*	*	*			

ORI #K,CCR
B

Operazione di OR a CCR
#K or CCR → CCR

ORI #K,SR
W

Operazione di OR a SR !Privileg.!
#K or SR → SR

PEA ea
L

Push indirizzo effettivo
D → -(SP)

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
D:			*			*	*	*	*	*	*	*

RESET

Resettaggio !Privileg.!
Reimposta la linea di Reset a 0 per
124 cicli di clock

ROL Dn.Dn / ROL #K,Dn / ROL ea
B W L

Rotazione a sinistra
D ruotato di n Bit → D

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
D:			*	*	*	*	*	*	*			

Il Bit MS va nel Bit LS e nel Flag Carry e sposta a sinistra

ROR Dn,Dn / ROR #K,Dn / ROR ea
B W L

Rotiere rechts
D ruotato di n Bit → D

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
D:			*	*	*	*	*	*	*			

Il Bit LS va nel Bit MS e nel Flag Carry e sposta a destra

ROXL Dn,Dn / ROXL #K,Dn / ROXL ea
B W L

Rototazione a sinistra con Flag X
D ruotato di n Bit → D

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
D:			*	*	*	*	*	*	*			

X va nel Bit LS e sposta a sinistra. Il Bit MS va in X e Carry

---

ROXR Dn,Dn / RORL #K,Dn / RORL ea — Rotazione a destra con Flag X
B W L — D ruotato di n Bit → D

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
D:			*	*	*	*	*	*	*			

X va nel Bit MS e sposta a destra. Il Bit LS va in X e Carry

RTE — Return da Eccezione !Privileg.!
(Sp) + → SR;(SP) + → PC

RTR — Return con Flag
(Sp) + → CCR;(SP) + → PC

RTS — Return
(SP) + → PC

SBCD Dn,Dn /ABCD -(An),-(An) — Sottrazione di BCD
B — D – S – X → D

Scc — Imposta il Byte se cc true
if cc = true $FF → Byte
B W L — else $00 → Byte

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
D:			*	*	*	*	*	*	*			

STOP # K — Caricamento di SR e Stop !Privileg.!
#K → SR; Fermo fino all'Interrupt

SUB ea,Dn / ADD Dn,ea — Sottrazione
B W L — D - S → D

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
S:	*	*	WL	*	*	*	*	*	*	*	*	*
D:			*	*	*	*	*	*	*			

SUBA ea,An — Sottrazione Indirizzo
W L — D - S → D

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
S:	*	WL	*	*	*	*	*	*	*	*	*	*

L’operando Parola viene ampliato in EXT.L

SUBI #K,ea — Sottrazione Costante
B W L — D - # K → D

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
D:	*		*	*	*	*	*	*	*			

SUBQ #K,ea — Sottrazione Costante Quick (#K < =8)
B W L — D - #K → D

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
D:	*	WL	*	*	*	*	*	*	*			

SUBX Dn,Dn / ADDX -(An),-(An) — Sottrazione con Flag X
D - S - X → D

SWAP Dn — Scambio Parole in Dn
W — Bit 31..16 < - > Bit 15..0

TAS ea
B

Prova e impostazione Bit **7** nel Byte
Bit7 → N/Z-Flag
1 → Bit **7**

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
D:	*		*	*	*	*	*	*	*			

TRAP #n

Trap-Exception
PC → -(SSP)
SR → -(SSP)
Vettore n → PC

TRAPV

Trap, se Overflow

TST ea
B W L

Prova operando rispetto a zero
Risultato in Flag N/Z

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
D:	*		*	*	*	*	*	*	*			

UNLK An

Unlink
An → SP; (SP) + → An

Significato dei Condition Code

	Abbreviazione	Significato	Spiegazione
	CC	Carry Clear	Carry = 0
	CS	Carry Set	Carry = 1
	EQ	Equal	Z =1
	GE	Greater or Equal	> =
***	GT	Greater Than	>
	HI	Higher	>
***	LE	Less or Equal	<
	LS	Less or Same	< =
***	LT	Less Than	<
	MI	Minus	-
	NE	Not Equal	< >
	PL	Plus	+
***	VC	oVerflow Clear	V = 0
***	VS	oVerflow Set	V =1

*** Per numeri preceduti da segno

Appendice A2: Library Vector Offset

Nella presente Appendice troviamo gli _LVO, e cioè
A2.1 Exec-Library
A2.2 DOS-Library
A23 Intuition-Library
A2.4 Graphics-Library
A2.5 Icon-Library
A2.6 Le tre Library matematiche
A2.7 Varie (Diskfont e Translator)

tutte in ordine alfabetico.

Nota: I nomi degli LVO corrispondono a quelli della documentazione Amiga. A tali nomi fanno anche riferimento gli Assembler della Metacomco e della HiSoft. Per quanto concerne l'Assembler SEKA sarà necessario tralasciare il trattino di sottolineatura prima delle lettere LVO e l'indirizzo di base. L'indirizzo di base per Exec è una costante, chiamata _SysBase equ 4.

Per tutte le altre Library la base è una variabile. Per l'HiSoft e SEKA se ne dovrà prevedere lo spazio nel programma con dc.l. Dopo l'apertura, d0 deve venire copiato in questa variabile se si vogliono utilizzare le macro standard. Nel caso del Metacomco è sufficiente XDEF (e linkare con amiga.lib).

A2.1 Exec-Library

Nome per l'apertura: `exec.library` (non serve mai)
Indirizzo di base: `_SysBase`

```
_LVOAbortIO             equ     -480
_LVOAddDevice           equ     -432
_LVOAddHead             equ     -240
_LVOAddIntServer        equ     -168
_LVOAddLibrary          equ     -396
_LVOAddPort             equ     -354
_LVOAddResource         equ     -486
_LVOAddTail             equ     -246
_LVOAddTask             equ     -282
_LVOAlert               equ     -108
_LVOAllocAbs            equ     -204
_LVOAllocate            equ     -186
_LVOAllocEntry          equ     -222
_LVOAllocMem            equ     -198
```

```
_LVOAllocTrap           equ     -342
_LVOAvailMem            equ     -216
_LVOCause               equ     -180
_LVOCheckIO             equ     -468
_LVOCloseDevice         equ     -450
_LVOCloseLibrary        equ     -414
_LVODeallocate          equ     -192
_LVODebug               equ     -114
_LVODisable             equ     -120
_LVODispatch            equ     -60
_LVODoIO                equ     -456
_LVOEnable              equ     -126
_LVOEnqueue             equ     -270
_LVOException           equ     -66
_LVOExitIntr            equ     -36
_LVOFindName            equ     -276
_LVOFindPort            equ     -390
_LVOFindResident        equ     -96
_LVOFindTask            equ     -294
_LVOForbid              equ     -132
_LVOFreeEntry           equ     -228
_LVOFreeMem             equ     -210
_LVOFreeSignal          equ     -336
_LVOFreeTrap            equ     -348
_LVOGetCC               equ     -528
_LVOGetHsg              equ     -372
_LVOInitCode            equ     -72
_LVOInitResident        equ     -102
_LVOInitStruct          equ     -78
_LVOInsert              equ     -234
_LVOMakeFunctions       equ     -90
_LVOMakeLibrary         equ     -84
_LVOOldOpenLibrary      equ     -408
_LVOOpenDevice          equ     -444
_LVOOpenLibrary         equ     -552
_LVOOpenResource        equ     -498
_LVOPermit              equ     -138
_LVOProcure             equ     -540
_LVOPutMsg              equ     -366
_LVORavDoFmt            equ     -522
_LVORawIOInit           equ     -504
_LVORavMayGetChar       equ     -510
_LVORavPutChar          equ     -516
_LVORemDevice           equ     -438
_LVORemHead             equ     -258
_LVORemIntServer        equ     -174
_LVORemLibrary          equ     -402
_LVORemove              equ     -252
_LVORemPort             equ     -360
_LVORemResource         equ     -492
_LVORemTail             equ     -264
_LVORemTask             equ     -288
```

```
_LVOReplyMsg            equ     -378
_LVOReschedule          equ     -48
_LVOSchedule            equ     -42
_LVOSendIO              equ     -462
_LVOSetExcept           equ     -312
_LVOSetFunction         equ     -420
_LVOSetIntVector        equ     -162
_LVOSetSignal           equ     -306
_LVOSetSR               equ     -144
_LVOSetTaskPri          equ     -300
_LVOSignal              equ     -324
_LVOAllocSignal         equ     -330
_LVOSumLibrary          equ     -426
_LVOSuperState          equ     -150
_LVOSupervisor          equ     -30
_LVOSwitch              equ     -54
_LVOTypeOfMem           equ     -534
_LVOUserState           equ     -156
_LVOVacate              equ     -546
_LVOWait                equ     -318
_LVOWaitIO              equ     -474
_LVOWaitPort            equ     -384
```

A2.2 DOS-Library

Nome per l'apertura: `dos.library`
Indirizzo di base: `_DOSBase`

```
_LVOClose               equ      -36
_LVOCreateDir           equ      -120
_LVOCreateProc          equ      -138
_LVOCurrentDir          equ      -126
_LVODateStamp           equ      -192
_LVODelay               equ      -198
_LVODeletefile          equ      -72
_LVODeviceProc          equ      -174
_LVODupLock             equ      -96
_LVOExamine             equ      -102
_LVOExecute             equ      -222
_LVOExit                equ      -144
_LVOExNext              equ      -108
_LVOGetPacket           equ      -162
_LVOInfo                equ      -114
_LVOInput               equ      -54
_LVOIoErr               equ      -132
_LVOIsInteractive       equ      -216
_LVOLoadSeg             equ      -150
_LVOLock                equ      -84
_LVOOpen                equ      -30
_LVOOutput              equ      -60
```

```
_LVOParentDir           equ     -210
_LVOQueuePacket         equ     -168
_LVORead                equ     -42
_LVORename              equ     -78
_LVOSeek                equ     -66
_LVOSetComment          equ     -180
_LVOSetProtection       equ     -186
_LVOUnLoadSeg           equ     -156
_LVOUnLock              equ     -90
_LVOWaitForChar         equ     -204
_LVOWrite               equ     -48
```

A2.3 Intuition-Library

Nome per l'apertura: `intuition.library`
Indirizzo di base: `_IntuitionBase`

```
_LVOAddGadget           equ     -42
_LVOAllocRemember       equ     -396
_LVOAlohaWorkbench      equ     -402
_LVOAutoRequest         equ     -348
_LVOBeginRefresh        equ     -354
_LVOBuildSysRequest     equ     -360
_LVOClearDMRequest      equ     -48
_LVOClearMenuStrip      equ     -54
_LVOClearPointer        equ     -60
_LVOCloseScreen         equ     -66
_LVOCloseWindow         equ     -72
_LVOCloseWorkBench      equ     -78
_LVOCurrentTime         equ     -84
_LVODisplayAlert        equ     -90
_LVODisplayBeep         equ     -96
_LVODoubleClick         equ     -102
_LVODrawBorder          equ     -108
_LVODravImage           equ     -114
_LVOEndRefresh          equ     -366
_LVOEndRequest          equ     -120
_LVOFreeRemember        equ     -408
_LVOFreeSysRequest      equ     -372
_LVOGetDefPrefs         equ     -126
_LVOGetPrefs            equ     -132
_LVOInitRequester       equ     -138
_LVOIntuition           equ     -36
_LVOItemAddress         equ     -144
_LVOLockIBase           equ     -414
_LVOMakeScreen          equ     -378
_LVOModifyIDCMP         equ     -150
_LVOModifyProp          equ     -156
_LVOMoveScreen          equ     -162
_LVOMoveWindov          equ     -168
```

```
_LVOOffGadget            equ      -174
_LVOOffMenu              equ      -180
_LVOOnGadget             equ      -186
_LVOOnMenu               equ      -192
_LVOOpenIntuition        equ      -30
_LVOOpenScreen           equ      -198
_LVOOpenWindow           equ      -204
_LVOOpenWorkBench        equ      -210
_LVOPrintIText           equ      -216
_LVORefreshGadgets       equ      -222
_LVORemakeDisplay        equ      -384
_LVORemoveGadget         equ      -228
_LVOReportMouse          equ      -234
_LVORequest              equ      -240
_LVORethinkDisplay       equ      -390
_LVOScreenToBack         equ      -246
_LVOScreenToFront        equ      -252
_LVOSetDMRequest         equ      -258
_LVOSetMenuStrip         equ      -264
_LVOSetPointer           equ      -270
_LVOSetPrefs             equ      -324
_LVOIntuiTextLength      equ      -330
_LVOSetWindowTitles      equ      -276
_LVOShowTitle            equ      -282
_LVOSizeWindow           equ      -288
_LVOUnlockIBase          equ      -420
_LVOViewAddress          equ      -294
_LVOViewPortAddress      equ      -300
_LVOWBenchToBack         equ      -336
_LVOWBenchToFront        equ      -342
_LVOWindowLimits         equ      -318
_LVOWindowToBack         equ      -306
_LVOWindowToFront        equ      -312
```

A2.4 Graphics-Library

Nome per l'apertura: `graphics.library`
Indirizzo di base: `_GfxBase`

```
_LVOAddAnimOb            equ     -156
_LVOAddBob               equ     -96
_LVOAddFont              equ     -480
_LVOAddVSprite           equ     -102
_LVOAllocRaster          equ     -492
_LVOAndRectRegion        equ     -504
_LVOAnimate              equ     -162
_LVOAreaDrav             equ     -258
_LVOAreaEnd              equ     -264
_LVOAreaMove             equ     -252
_LVOAskFont              equ     -474
```

```
_LVOAskSoftStyle        equ     -84
_LVOBltBitMap           equ     -30
_LVOBltBitMapRastPort   equ     -606
_LVOBltClear            equ     -300
_LVOBltPattern          equ     -312
_LVOBltTemplate         equ     -36
_LVOCBump               equ     -366
_LVOChangeSprite        equ     -420
_LVOClearEOL            equ     -42
_LVOClearRegion         equ     -528
_LVOClearScreen         equ     -48
_LVOClipBlit            equ     -552
_LVOCloseFont           equ     -78
_LVOCMove               equ     -372
_LVOCopySBitMap         equ     -450
_LVOCWait               equ     -378
_LVODisownBlitter       equ     -462
_LVODisposeRegion       equ     -534
_LVODoCollision         equ     -108
_LVODraw                equ     -246
_LVODrawGList           equ     -114
_LVOFlood               equ     -330
_LVOFreeColorMap        equ     -576
_LVOFreeCopList         equ     -546
_LVOFreeCprList         equ     -564
_LVOFreeGBuffers        equ     -600
_LVOFreeRaster          equ     -498
_LVOFreeSprite          equ     -414
_LVOFreeVPortCopLists   equ     -540
_LVOGelsFuncE           equ     -180
_LVOGelsFuncF           equ     -186
_LVOGetColorMap         equ     -570
_LVOGetGBuffers         equ     -168
_LVOGetRGB4             equ     -582
_LVOGetSprite           equ     -408
_LVOInitArea            equ     -282
_LVOInitBitMap          equ     -390
_LVOInitGels            equ     -120
_LVOInitGMasks          equ     -174
_LVOInitMasks           equ     -126
_LVOInitRastPort        equ     -198
_LVOInitTmpRas          equ     -468
_LVOInitView            equ     -360
_LVOInitVPort           equ     -204
_LVOLoadRGB4            equ     -192
_LVOLoadView            equ     -222
_LVOLockLayerRom        equ     -432
_LVOMakeVPort           equ     -216
_LVOMove                equ     -240
_LVOMoveSprite          equ     -426
_LVOMrgCop              equ     -210
_LVONewRegion           equ     -516
```

```
_LVONotRegion           equ     -522
_LVOOpenFont            equ     -72
_LVOCrRectRegion        equ     -510
_LVOOwnBlItter          equ     -456
_LVOPolyDraw            equ     -336
_LVOQBlit               equ     -276
_LVOQBSBlit             equ     -294
_LVOReadPixel           equ     -318
_LVORectFIll            equ     -306
_LVORemFont             equ     -486
_LVORemIBob             equ     -132
_LVORemVSprite          equ     -138
_LVOScrollRaster        equ     -396
_LVOScrollVPort         equ     -588
_LVOSetAPen             equ     -342
_LVOSetBPen             equ     -348
_LVOSetCollision        equ     -144
_LVOSetDrMd             equ     -354
_LVOSetFont             equ     -66
_LVOSetRast             equ     -234
_LVOSetRGB4             equ     -288
_LVOSetSoftStyle        equ      -90
_LVOSortGList           equ     -150
_LVOSyncSBitHap         equ     -444
_LVOText                equ     -60
_LVOTextLength          equ     -54
_LVOUCopperListInit     equ     -594
_LVOUnlockLayerRom      equ     -438
_LVOVBeamPos            equ     -384
_LVOWaitBlit            equ     -228
_LVOWaitBOVP            equ     -402
_LVOWaitTOF             equ     -270
_LVOWritePixel          equ     -324
_LVOXorRectRegion       equ     -558
```

A2.5 Icon-Library

Nome per l'apertura: `icon.library`
Indirizzo di base: `_IconBase`

```
_LVOAddFreeList         equ     -72
_LVOAllocWBObject       equ     -66
_LVOBumpRevision        equ     -108
_LVOFindToolType        equ     -96
_LVOFreeDiskObject      equ     -90
_LVOFreeFreeList        equ     -54
_LVOFreeWBObject        equ     -60
_LVOGetDiskObject       equ     -78
_LVOGetIcon             equ     -42
_LVOGetWB0bject         equ     -30
```

```
_LVOMatchToolValue         equ      -102
_LVOPutDiskObject          equ      -84
_LVOPutIcon                equ      -48
_LVOPutWBObject            equ      -36
```

A2.6 Le librerie matematiche

Nome per l'apertura: `mathffp.library`
Indirizzo di base: `_MathBase`

```
_LVOSPAbs                  equ      -54
_LVOSPAdd                  equ      -66
_LVOSPCmp                  equ      -42
_LVOSPDiv                  equ      -84
_LVOSPFix                  equ      -30
_LVOSPFH                   equ      -36
_LVOSPMul                  equ      -78
_LVOSPNeg                  equ      -60
_LVOSPSub                  equ      -72
_LVOSPTst                  equ      -48
```

Nome per l'apertura: `mathieeedoubbas.library`
Indirizzo di base: `_MathleeeDoubBasBase`

```
_LVOIEEEDPAbs              equ      -54
_LVOIEEEDPAdd              equ      -66
_LVOIEEEDPCmp              equ      -42
_LVOIEEEDPDiv              equ      -84
_LVOIEEEDPFix              equ      -30
_LVOIEEEDPFlt              equ      -36
_LVOIEEEDPMul              equ      -78
_LVOIEEEDPNeg              equ      -60
_LVOIEEEDPSub              equ      -72
_LVOIEEEDPTst              equ      -48
```

Nome per l'apertura: `mathtrans.library`
Indirizzo di base: `_MathTransBase`

```
_LVOSPAcos                 equ      -120
_LVOSPAsin                 equ      -114
_LVOSPAtan                 equ      -30
_LVOSPCos                  equ      -42
_LVOSPCosh                 equ      -66
_LVOSPExp                  equ      -78
_LVOSPFieee                equ      -108
_LVOSPLog                  equ      -84
```

```
_LVOSPLog10              equ      -126
_LVOSPPow                equ      -90
_LVOSPSin                equ      -36
_LVOSPSincos             equ      -54
_LVOSPSinh               equ      -60
_LVOSPSqrt               equ      -96
_LVOSPTan                equ      -48
_LVOSPTanh               equ      -72
_LVOSPTieee              equ      -102
```

A2.7 Varie (Diskfont e Translator)

Diskfont-Library
Nome per l'apertura: `diskfont.library`
Indirizzo di base: `DiskfontBase`

```
_LVOAvaIlFonts           equ      -36
_LVOOpenDiskFont         equ      -30
```

Translator-Library
Nome per l'apertura: `translator.library`
Indirizzo di base: `TranslatorBase`

```
LVOTranslate             equ       -30
```

Appendice A3: Funzioni più importanti e loro parametri

A3.1 Exec

Funzione/Reg.	A0	A1	A2	A3	D0	D1
AbortIO		IORequest				
AddDevice		Device				
AddHead	List	Node				
AddIntServer		Interrupt			IntNumber	
AddLibrary		Library				
AddPort		Port				
AddResource		Resource				
AddTail		List	Node			
AddTask		Task	initPC	finalPC		
AllocAbs		LocatIOn			Size	
Allocate	freeList				Size	
AllocEntry	Entry					
AllocMem					Size	Request
AllocSignal					SignalNum	
AllocTrap					TrapNum	
AvailMem						Request
Cause		Interrupt				
CheckIO		IORequest				
CloseDevice		IORequest				
CloseLibrary		Library				
Deallocate		freeList	memoryBlock		Size	
Disable						
DoIO		IORequest				
Enable						
Enqueue	List	Node				
FindName	List	name				
FindPort		name				

Funzione/Reg.	A0	A1	A2	A3	D0	D1
FindTask		name				
Forbid						
FreeEntry	Entry					
FreeMem		MemoryBlock			Size	
FreeSignal					SignalNum	
FreeTrap					TrapNum	
GetMsg	Port					
Insert	List	Node	pred			
OldOpenLibrary		LibName				
OpenDevice	devName	IORequest			unit	Flags
OpenResource	resName				Version	
Permit						
PutMsg	Port	message				
RemDevice		Device				
RemHead	List					
RemlntServer		Interrupt			IntNumber	
RemLibrary		Library				
Remove		Node				
RemPort		Port				
RemResource		Resource				
RemTail	List					
RemTask		Task				
ReplyMsg		Message				
SendIO		IORequest				
SetExcept					NewSig	SignalSet
SetlntVector		Interrupt			IntNum	Interrupt
SetSignal					NewSig	SignalSet
SetSR					NewSR	Mask
SetTaskPri		Task			Priority	
Signal		Task			SignalSet	
SumLibrary		Library				
SuperState						
UserState					SysStack	
Wait					SignalSet	
WaitIO		IORequest				
WaitPort	Port					

A3.2 DOS

Funzione/Reg.	D1	D2	D3	D4
Close	File			
CreateDir	Name			
CreateProc	Name	Prior.	SegList	StackSize
CurrentDir	Lock			
DateStamp	Date			
Delay	Timeout			
DeleteFil	Name			
DeviceProc	Name			
DupLock	Lock			
Examine	Lock	FileInfoBLock		
Execute	String	File	File	
Exit	ReturnCode			
ExNext	Lock	FileInfoBLock		
Info	Lock	ParameterBLock		
Input				
IoErr				
IsInteractive	File			
LoadSeg	FileName			
Lock	Name	Type		
Open	Name	AccessMode		
Output				
ParentDir	Lock			
Read	File	Buffer	Len	
ReName	OldName	NewName		
Seek	File	Position	Offset	
SetComment	Name	Comment		
SetProtection	Name	Mask		
UnLoadSeg	Segment			
UnLock	Lock			
WaitForChar	File	Timeout		
Write	File	Buffer	Len	

A3.3 Intuition

AddGadget	AddPtr/Gadget/Position A0/A1/D0
AllocRemember	RememberKey/Sizef/Flags A0/D0/D1
AlohaWorkbench	wbport A0
AutoRequest	Window/Body/PText/NText/PFlag/NFlag/W/H A0 / A 1 /A2 /A3 /D0 / D 1 /D2/D3
BeginRefresh	Window A0

BuildSysRequest	Window/Body/PosText/NegText/Flags/W/H
	A0 / A1 /A2 /A3 /D0 /D1/D2
ClearDMRequest	Window A0
ClearMenuStrip	Window A0
ClearPointer	Window A0
CloseScreen	Screen A0
CloseWindow	Window A0
CloseWorkBench	
CurrentTime	Seconds/Micros A0/A1
DisplayAlert	AlertNumber/String/Height D0/A0/D1
DisplayBeep	Screen A0
DoubleClick	sseconds/smicros/cseconds/cmicros D0/D 1/D2/D3
DrawBorder	RPort/Border/LeftOffset/TopOffset A0/A1/D0/D1
Drawlmage	RPort/Image/LeftOffset/TopOffset A0/A1/D0/D1
EndRefresh	Window/complete A0/D0
EndRequest	requester/Window A0/A1
FreeRemember	RememberKey/ReallyForget A0/D0
FreeSysRequest	Window A0
GetDefPrefs	Preferences/Size A0/D0
GetPrefs	Preferences/Size A0/D0
InitRequester	req A0
IntuiTextLength	itext A0
Intuition	ievent A0
ItemAddress	MenuStrip/MenuNumber A0/D0
MakeScreen	Screen A0
ModifylDCMP	Window/Flags A0/D0
ModifyProp	Gadget/Ptr/Req/Flags/HPos/VPos/HBody/VBody
	A0 /A1/A2 /D0 / D 1 /D2 /D3 /D4
MoveScreen	Screen/dx/dy A0/D0/D1
MoveWindow	Window/dx/dy A0/D0/D1
OffGadget	Gadget/Ptr/Req A0/A1/A2
OffMenu	Window/MenuNumber A0/D0
OnGadget	Gadget/Ptr/Req A0/A1/A2
OnMenu	Window/MenuNumber A0/D0
Openlntuition	
OpenScreen	OSargs A0
OpenWindow	OWargs A0
OpenWorkBench	
PrintIText	rp/itext/left/top A0/A1/D0/D1
RefreshGadgets	Gadgets/Ptr/Req A0/A1/A2
RemakeDisplay	
RemoveGadget	RemPtr/Gadget A0/A1
ReportMouse	Window/Boolean A0/D0

Request	Requester/Window A0/A1
RethinkDisplay	
ScreenToBack	Screen A0
ScreenToFront	Screen A0
SetDMRequest	Window/req A0/A1
SetMenuStrip	Window/Menu A0/A1
SetPointer	Window/Pointer/Height/Width/Xoffset/Yoffset
	A0 /A1 /D0 /D 1 /D2 /D3
SetPrefs	Preferences/Size/flag A0/D0/D1
SetWindowTitles	Window/Windowtitle/Screentitle A0/A1/A2
ShowTitle	Screen/Showlt A0/D0
Size Window	Window/dx/dy A0/D0/D1
ViewAddress	
ViewPortAddress	Window A0
WBenchToBack	
WBenchToFront	
WindowLimits	Window/minwidth / minheight / maxwidth / maxheight
	A0 /D0 / D 1 /D2 /D3
WindowToBack	Window A0
WindowToFront	Window A0

A3.4 Graphics

AddAnimOb	obj/animationKey/rastPort A0/A1/A2
AddBob	bob/rastPort A0/A1
AddFont	textFont A1
AddVSprite	vSprite/rastPort A0/A1
AllocRaster	width/height D0/D1
AndRectRegion	rgn/rect A0/A1
Animate	animationKey/rastPort A0/A1
AreaDraw	rastPort/x/y A1/D0/D1
AreaEnd	rastPort A1
AreaMove	rastPort/x/y A1/D0/D1
AskFont	rastPort/textAttr A1/A0
AskSoftStyle	rastPort A1
BltBitMap	srcBitMap/srcX/srcY/destBitMap/destX/destY
	A0 /D0 /D 1 /A1 /D2 /D3
	sizeX/sizeY/minterm/mask/tempA
	D4 /D5 /D6 /D7 /A2
BltBitMapRastPort	srcbm/srcx/ srcy/destrp/destX/destY
	A0 /D0 /Dl /A1 /D2 /D3
	sizeX / size Y / minterm
	D4 /D5 /D6

BltClear	memory/size/flags A1/D0/D1
BltPattern	rastPort/ras/xl/yl/maxX/maxY / fillBytes A1 /A0/D0/D1/D2 /D3 /D4
BltTemplate	src/srcX/srcMod / destRastPort / destX/destY/sizeX/sizeY A0/D0/D1 /A1 /D2 /D3 /D4 /D5
CBump	copperList A1
ChangeSprite	vp/simplesprite/data A0/A1/A2
ClearEOL	rastPort A1
ClearRegion	rgn A0
ClearScreen	rastPort A1
ClipBlit	src/srcX/srcY/destrp/destX/destY/sizeX/sizeY/minterm A0/D0 /Dl /A1 /D2 /D3 /D4 /D5 /D6
CloseFont	textFont A1
CMove	copperList/destination/data A1/D0/D1
CopySBitMap	I1/I2 A0/A1
CWait	copperList/x/y A1/D0/D1
DisownBlitter	
DisposeRegion	rgn A0
DoCollision	rasPort A1
Draw	rastPort/x/y A1/D0/D1
DrawGList	rastPort/viewPort A1/A0
Flood	rastPort/mode/x/y A1/D2/D0/D1
FreeColorMap	colormap A0
FreeCopList	coplist A0
FreeCprList	cprlist A0
FreeGBuffers	animationObj/rastPort/doubleBuffer A0/A1/D0
FreeRaster	planeptr/width/height A0/D0/D1
FreeSprite	num D0
FreeVPortCopLists	viewport A0
GelsFuncE	
GelsFuncF	
GetColorMap	entries D0
GetGBuffers	animationObj/rastPort/doubleBuffer A0/A1/D0
GetRGB4	colormap/entry A0/D0
GetSprite	simplesprite/num A0/D0
InitArea	arealnfo/vectorTable/vectorTableSize A0/A1/D0
InitBitMap	bitMap/depth/width/height A0/D0/D1 /D2
InitGels	dummyHead/dummyTail/Gelsinfo A0/AI/A2
InitGMasks	animationObj A0
InitMasks	vSprite A0
InitRastPort	rastPort A1
InitTmpRas	tmpras/buff/size A0/A1/D0
InitView	view A1

InitVPort	viewPort A0
LoadRGB4	viewPort/colors/count A0/A1/D0
LoadView	view A1
LockLayerRom	layer A5
MakeVPort	view/viewPort A0/A1
Move	rastPort/x/y A1/D0/D1
MoveSprite	viewport/simplesprite/x/y A0/A1/D0/D1
MrgCop	view A1
NewRegion	
NotRegion	rgn A0
OpenFont	textAttr A0
OrRectRegion	rgn/rect A0/A1
OwnBlitter	
PolyDraw	rastPort/count/polyTable A1/D0/A0
QBlit	blit A1
QBSBlit	blit A1
ReadPixel	rastPort/x/y A1/D0/D1
RectFill	rastPort/xl/yl/xu/yu A1/D0/D1/D2/D3
RemFont	textFont A1
RemIBob	bob/rastPort/viewPort A0/A1/A2
RemVSprite	vSprite A0
ScrollRaster	rastPort/dX/dY/minx/miny/maxx/maxy
	A1 /D0/D1/D2 /D3 /D4 /D5
ScrollVPort	vp A0
SetAPen	rastPort/pen A1/D0
SetBPen	rastPort/pen A1/D0
SetCollision	type/routine/gelslnfo D0/A0/A1
SetDrMd	rastPort/drawMode A1/D0
SetFont	RastPortID/textFont A1/A0
SetRast	rastPort/color A1/D0
SetRGB4	viewPort/index/r/g/b A0/D0/D1/D2/D3
SetSoftStyle	rastPort/style/enable A1/D0/D1
SortGList	rastPort A1
SyncSBitMap	I A0
Text	RastPort/string/count A1/A0/D0
TextLength	RastPort/string/count A1/A0/D0
UCopperListlnit	copperlist/num A0/D0
UnlockLayerRom	layer A5
VBeamPos	
WaitBlit	
WaitBOVP	viewport A0
WaitTOF	

WritePixel	rastPort/x/y A1/D0/D1
XorRectRegion	rgn/rect A0/A1

A3.5 Layers (li sta per layer info)

BeginUpdate	Layer A0
BehindLayer	li/Layer A0/A1
CreateBehindLayer	li/bm/x0/y0/xl/yl/flags/bm2 A0/A1/D0/D1/D2/D3/D4/A2
CreateUpfrontLayer	h/bm/x0/y0/xl/yl/flags/bm2 A0/A1/D0/D1/D2/D3/D4/A2
DeleteLayer	li/Layer A0/A1
DisposeLayerlnfo	li A0
EndUpdate	Layer/flag A0/D0
FattenLayerlnfo	li A0
InitLayers	li A0
LockLayer	li/Layer A0/A1
LockLayerlnfo	li A0
LockLayers	li A0
MoveLayer	li/Layer/dx/dy A0/A1/D0/D1
MoveLayerlnFrontOf	Layer/Layer A0/A1
NewLayerlnfo	
ScrollLayer	li/Layer/dx/dy A0/A1/D0/D1
SizeLayer	li/Layer/dx/dy A0/A1/D0/D1
SwapBitsRastPortClip-Rect	rp/cr A0/A1
ThinLayerlnfo	li A0
UnlockLayer	Layer A0
UnlockLayerlnfo	li A0
UnlockLayers	li A0
UpfrontLayer	li/Layer A0/A1
WhichLayer	li/x/y A0/D0/D1

Appendice A4: Tipi di dati, strutture, tabelle di offset, costanti

A4.1 Exec
A4.2 DOS
A4.3 Intuition
A4.4 Graphics
A45 Devices

La presente appendice è costituita in modo tale che parti di essa, (o tutta) possano venire facilmente trasformate in file Include. Dal momento che non voglio costringere nessuno ad usare sempre la soluzione DC oppure la soluzione di Offset, a seconda del fatto che io ritenga più opportuna l'una o l'altra, stabiliamo il seguente compromesso: tutte le strutture sono costituite con DS.x. Nella prima colonna è tuttavia indicato, l'Offset. L'ultima riga fornisce la dimensione tramite una istruzione EQU.

Gli Offset della prima colonna sono sempre annotati in esadecimale. Anche per gli operatori, vale l'esadecimale come sempre ($ prima del numero). La riga seguente ne fornisce un esempio:

```
1A      nw_Title        ds.l    1
```

Da ciò potremmo trarre:

```
        dc.l    Mio_Titolo
```

oppure (xx _SIZE è sempre definito come EQU):

```
        NewWindow       ds.b            nw_SIZE
        nw_Title        equ             $1A
                        lea             NewWindow,a0
                        move.l          #Mio_Titolo,nw_Title(a0)
```

Le costanti appartenenti ad una struttura si trovano immediatamente dopo queste righe, e di solito sotto forma di istruzioni EQU. Se due strutture utilizzano le stesse costanti, queste si troveranno fra le due strutture. In casi particolari troveremo delle costanti anche in altre situazioni.

Ulteriori dettagli per il funzionamento delle strutture e delle tabelle di Offset sono contenuti nel Capitolo 11. L'accento circonflesso (^) significa “puntatore a” (indirizzo di).

A4.1 Exec

```
;nodes
;-----
000              LN_SUCC  ds.l     1
004              LN_PRED  ds.l     1
008              LN_TYPE  ds.b     1
009              LN_PRI   ds.b     1
00A              LN_NAME  ds.l     1
                 LN_SIZE  equ      $00E

NT_UNKNOWN       equ      0
NT_TASK          equ      1
NT_INTERRUPT     equ      2
NT_DEVICE        equ      3
NT_MSGPORT       equ      4
NT_MESSAGE       equ      5
NT_FREEMSG       equ      6
NT_REPLYMSG      equ      7
NT_RESOURCE      equ      8
NT_LIBRARY       equ      9
NT_MEMORY        equ      10
NT_SOFTINT       equ      11
NT_FONT          equ      12
NT_PROCESS       equ      13
NT SEMAPHORE     equ      14

;La Base di sys conta a partire da qui
;---------------------

; lists
;------

000      LH_HEAD          ds.l     1
004      LH_TAIL          ds.l     1
008      LH_TAILPRED      ds.l     1
00C      LH_TYPE          ds.b     1
00D      LH_pad           ds.b     1
         LH_SIZE          equ      $00E

;Struttura della Message-Port
;---------------------

00E      MP_FLAGS         ds.b     1        ;Flags
00F      MP_SIGBIT        ds.b     1        ;Numero Bit Segnale
010      MP_SIGTASK       ds.l     1        ;^Task per il Sig
014      MP_MSGLIST       ds.b     LH_SIZE  ;Lista dei Messaggi
         MP SIZE          equ      $022

;Struttura di Message
00E      MN_REPLYPORT     ds.l     1        ;^Reply Port
012      MN_LENGTH        ds.w     1        ;Lunghezza in Byte
```

```
014     MN_SIZE          equ     $014

MP_SOFTINT       equ     MP_SIGTASK
PF_ACTION        equ     3

PA_SIGNAL        equ     0
PA_SOFTINT       equ     1
PA_IGNORE        equ     2


;Libraries
;---------

LIB_VECTSIZE     equ     6
LIB_RESERVED     equ     4
LIB_BASE         equ     $FFFFFFFA
LIB_USERDEF      equ     LIB_BASE-(LIB_RESERVED*LIB_VECTSIZE)
LIB_NONSTD       equ     LIB_USERDEF

00E     LIB_FLAGS        ds.b    1        ;Per i Flag vedi sotto
00F     LIB_pad          ds.b    1        ;
010     LIB_NEGSIZE      ds.w    1        ;Numero Byte prima di Lib
012     LIB_POSSIZE      ds.w    1        ;Numero Byte dopo Lib
014     LIB_VERSION      ds.w    1        ;Versione (principale)
016     LIB_REVISION     ds.w    1        ;Versione sottogruppo
018     LIB_IDSTRING     ds.l    1        ;^Name
01C     LIB_SUM          ds.l    1        ;Checksum
020     LIB_OPENCNT      ds.w    1        ;Open attuali
        LIB_SIZE         equ     $022


LIBB_SUMMING     equ     0       ;Calcolo della Checksum
LIBF_SUMMING     equ     1       ;
LIBB_CHANGED     equ     1       ;La Lib è stata modificata
LIBF_CHANGED     equ     2       ;
LIBB_SUMUSED     equ     2       ;1 se problema di Checksum
LIBF_SUMUSED     equ     4
LIBB_DELEXP      equ     3
LIBF_DELEXP      equ     8

;Semaphore Message Port
;----------------------
022     SM_BIDS          ds.w    1        ;Numero dei Lock Bit
        SM_SIZE          equ     $024

;Unions
;------
SM_LOCKMSG       equ     MP_SIGTASK       ;Ved. la’

;Struttura di Device
;-------------------
;************ come Library con DD_xxxx **********
DD_SIZE          equ     $22
```

```
022     UNIT_FLAGS       ds.b     1
023     UNIT_pad         ds.b     1
024     UNIT_OPENCNT     ds.w     1
        UNIT_SIZE        equ      $022

UNITB_ACTIVE     equ      0

;INTERRUPTS
;----------

00E     IS_DATA          ds.l     1
012     IS_CODE          ds.l     1
        IS_SIZE          equ      $016

000     IV_DATA          ds.l     1
004     IV_CODE          ds.l     1
008     IV_NODE          ds.l     1
        IV_SIZE          equ      $00C

SB_SAR           equ      15
SF_SAR           equ      $8000
SB_TQE           equ      14
SF_TQE           equ      $4000
SB_SINT          equ      13
SF_SINT          equ      $2000

SH_PAD           equ      SH_SIZE

SIH_PRIMASK      equ      $0F0
SIH_QUEUES       equ      5

;Variabili statistiche di sistema
;--------------------------------

022     SoftVer          ds.w     1        ;Versione Kickstart
024     LowMemChkSum     ds.w     1        ;Vettore trap Checksumm
026     ChkBase          ds.l     1        ;Base di Sys(complemento)
02A     ColdCapture      ds.l     1        ;^Cold-Start
02E     CoolCapture      ds.l     1        ;^Cool-Start
032     WarmCapture      ds.l     1        ;^Warm-Start
036     SysStkUpper      ds.l     1        ;^Sys-Stack (alto)
03A     SysStkLower      ds.l     1        ;^Sys-Stack (Top)
03E     MaxLocHem        ds.l     1        ;^Memoria max attuale
042     DebugEntry       ds.l     1        ;^Debugger
046     DebugData        ds.l     1        ;^Segmento Data Debugger
04A     AlertData        ds.l     1        ;^Alerts Data
04E     RsvdExt          ds.l     1        ;riservato
052     ChkSum           ds.w     1        ;Checksumm fino a qui
;Per sorgenti di Interrupt:
054     IntVects         equ      $054
054     IVTBE            ds.b     IV_SIZE
060     IVDSKBLK         ds.b     IV_SIZE
```

```
06C      IVSOFTINT        ds.b     IV_SIZE
078      IVPORTS          ds.b     IV_SIZE
084      IVCOPER          ds.b     IV_SIZE
090      IVVERTB          ds.b     IV_SIZE
09C      IVBLIT           ds.b     IV_SIZE
0A8      IVAUD0           ds.b     IV_SIZE
0B4      IVAUD1           ds.b     IV_SIZE
0C0      IVAUD2           ds.b     IV_SIZE
0CC      IVAUD3           ds.b     IV_SIZE
0D8      IVRBF            ds.b     IV_SIZE
0E4      IVDSKSYNC        ds.b     IV_SIZE
0F0      IVEXTER          ds.b     IV_SIZE
0FC      IVINTEN          ds.b     IV_SIZE
108      IVNMI            ds.b     IV_SIZE

;Variabili dinamiche di sistema
114      ThisTask         ds.l     1        ;^Task attuale
118      IdleCount        ds.l     1        ;Contatore di attesa
11C      DispCount        ds.l     1        ;Contatore di Dispatch
120      Quantum          ds.w     1        ;Quanto di tempo
122      Elapsed          ds.w     1        ;Trascorso
124      SysFlags         ds.w     1        ;vari
126      IDNestCnt        ds.b     1        ;Profondita' di Inter.-Disable
127      TDNestCnt        ds.b     1        ;Profondita' di Task-Disable

128      AttnFlags        ds.w     1        ;Flag del marcatore
12A      AttnResched      ds.w     1        ;
12C      ResModules       ds.l     1        ;^Moduli residenti

130      TaskTrapCode     ds.l     1        ;^Default Trap-Routine
134      TaskExceptCode   ds.l     1        ;^Default Exception-Rout.
138      TaskExitCode     ds.l     1        ;^Default Exit-Routine
13C      TaskSigAlloc     ds.l     1        ;Default Signal-Mask
140      TaskTrapAlloc    ds.w     1        ;Default Trap-Mask

;List Headers
142      MemList          ds.b     LH_SIZE
150      ResourceList     ds.b     LH_SIZE
15E      DeviceList       ds.b     LH_SIZE
16C      IntrList         ds.b     LH_SIZE
17A      LibList          ds.b     LH_SIZE
188      PortList         ds.b     LH_SIZE
196      TaskReady        ds.b     LH_SIZE
1A4      TaskWait         ds.b     LH_SIZE

1B2      Softlnts         ds.b     5*SH_SIZE
202      LastAlert        ds.b     16
212      ExecBaseReserved ds.l     1

SYSBASESIZE      equ      $216
;attention flags:
AFB_68010        equ      0        ; (anche 68020)
AFB_68020        equ      1
AFB_68881        equ      4
```

```
AFB_PAL          equ     8       ;PAL/NTSC
AFB_50HZ         equ     9       ;Clock-Rate

UNITF_ACTIVE     equ     1
UNITE_INTASK     equ     1
UNITF_INTASK     equ     2

;memory
;------

00E      ML_NUMENTRIES    ds.w    1
         ML_ME            equ     16
         ML_SIZE          equ     16

000      ME_REQS          ds.w    0
000      ME_ADDR          ds.l    1
004      ME_LENGTH        ds.l    1
         ME_SIZE          equ     8

MEMB_PUBLIC      equ     0
MEMF_PUBLIC      equ     1

MEMB_CHIP        equ     1
MEMF_CHIP        equ     2

MEMB_FAST        equ     2
MEMF_FAST        equ     4

MEMB_CLEAR       equ     16
MEMF_CLEAR       equ     $10000

MEMB_LARGEST     equ     17
MEMF_LARGEST     equ     $20000

MEM_BLOCKSIZE    equ     8
MEM_BLOCKMASK    equ     (MEM_BLOCKSIZE-1)

00E      MH_ATTRIBUTES    ds.w    1
010      MH_FIRST         ds.l    1
014      MH_LOWER         ds.l    1
018      MH_UPPER         ds.l    1
01C      MH_FREE          ds.l    1
         MH_SIZE          equ     $020

000      MC_NEXT          ds.l    1
004      MC_BYTES         ds.l    1
008      MC_SIZE          ds.l    1

;Struttura di Task Control
;-------------------------
00E      TC_JLAGS         ds.b    1       ;Flags
00F      TC_JTATE         ds.b    1       ;Status
010      TC_IDNESTCNT     ds.b    1       ;Profondita' di Inter.-Disable
```

```
011      TC_TDNESTCNT     ds.b     1        ;Profondita’ di Task-Disable
012      TC_SIGALLOC      ds.l     1        ;Segnali attribuiti
016      TC_SIGWAIT       ds.l     1        ;che sono attesi
01A      TC_SIGRECVD      ds.l     1        ;che arrivano
01E      TC_SIGEXCEPT     ds.l     1        ;che valgono come Exception
022      TC_TRAPALLOC     ds.w     1        ;Trap attribuiti
024      TC_TRAPABLE      ds.w     1        ;Trap abilitati
026      TC_EXCEPTDATA    ds.l     1        ;^Dati di Exception
02A      TC_EXCEPTCODE    ds.l     1        ;^Codice di Exception
02E      TC_TRAPDATA      ds.l     1        ;^Dati di Trap
032      TC_TRAPCODE      ds.l     1        ;^Codice di Trap
036      TC_SPREG         ds.l     1        ;Puntatore Stack
03A      TC_SPLOWER       ds.l     1        ;^Struttura Stack basso
03E      TC_SPUPPER       ds.l     1        ;^Struttura Stack alto (+2)
042      TC_SWITCH        ds.l     1        ;^Task che lascia la CPU
046      TC_LAUNCH        ds.l     1        ;^Task in arrivo
04A      TC_MEMENTRY      ds.b     LH_SIZE  ;Uso della memoria
058      TC_Userdata      ds.l     1        ;^Dati utente
         TC_SIZE          equ      $05C


TB_PROCTIME      equ      0
TF_PROCTIME      equ      1
TB_STACKCHK      equ      4
TF_STACKCHK      equ      16
TB_EXCEPT        equ      5
TF_EXCEPT        equ      52
TB_SWITCH        equ      6
TF_SWITCH        equ      64
TB_LAUNCH        equ      7
TF_LAUNCH        equ      128

TS_INVALID       equ      0
TS_ADDED         equ      TS_INVALID+1
TS_RUN           equ      TS_ADDED+1
TS_READY         equ      TS_RUN+1
TS_WAIT          equ      TS_READY+1
TS_EXCEPT        equ      TS_WAIT+1
TS_REMOVED       equ      TS_EXCEPT+1

SIGF_ABORT       equ      $0001
SIGF_CHILD       equ      $0002
SIGF_BLIT        equ      $0010
SIGF_DOS         equ      $0100

SIGB ABORT       equ      0
SIGB_CHILD       equ      1
SIGB_BLIT        equ      4
SIGB_DOS         equ      8

SYS_SIGALLOC     equ      $0FFFF
SYS_TRAPALLOC    equ      $08000
```

```
;Struttura di IO-Request
;-----------------------

014     IO_DEVICE        ds.l    1        ;^Struttura Device
018     IO_UNIT          ds.l    1        ;^Unit (dei Driver)
01C     IO_COMMAND       ds.w    1        ;^Comando Device
01E     IO_FLAGS         ds.b    1        ;Flag
01F     IO_ERROR         ds.b    1        ;Warning-Code
        IO_SIZE          equ     $020

;IO-Extension
020     IO_ACTUAL        ds.l    1        ;Byte trasmessi
024     IO_LENGTH        ds.l    1        ;Totale Byte
028     IO_DATA          ds.l    1        ;^Dati
02C     IO_OFFSET        ds.l    1        ;Offset di Seeking
        IOSTD_SIZE       equ     $30

IOB_QUICK equ 0
IOF_QUICK equ 1
```

A4.2 DOS

```
;Accesso ai File
;---------------
MODE_READWRITE   equ     1004    ;Solo dalla V.1.2
MODE_OLDFILE     equ     1005
MODE_READONLY    equ     MODE_OLDFILE
MODE_NEWFILE     equ     1006

;Funzione di SEEK, posizioni relative
OFFSET_BEGINNING equ     -1               ;Inizio del file
OFFSET_BEGINING  equ     OFFSET_BEGINNING
OFFSET_CURRENT   equ     0                ;relativo a quello impostato
OFFSET_END       equ     1                ;Fine del file

;banalità
;--------
BITSPERBYTE      equ     8
BYTESPERLONG     equ     4
BITSPERLONG      equ     52
MAXINT           equ     $7FFFFFFF
MININT           equ     $80000000

;Tipi di Lock
;------------
SHARED_LOCK      equ     -2               ;altri Task possono leggere
ACCESS_READ      equ     SHARED_LOCK
EXCLUSIVE_LOCK   equ     -1               ;non possono leggere
ACCESS_WRITE     equ     EXCLUSIVE_LOCK
```

```
;DateStamp
;---------
00      ds_Days         ds.l    1       ;Giorni dal 1.1.1978
04      ds_Minute       ds.l    1       ;Minuti dalle ore 00
08      ds_Tick         ds.l    1       ;Tick nel minuto corrente
        ds_SIZEOF       equ     $0C

TICKS_PER_SECOND        equ     50      ;1 Tick = 1/50 sec



;FileInfoBlock
;-------------
00      fib_DiskKey      ds.l    1                ;0=file, >0 = Dir
04      fib_DirEntryType ds.l    1                ;0=file, >0 = Dir
08      fib_FileName     ds.b    108              ;max. lunghezza 30
74      fib_Protection   ds.l    1                ;Vedi equ sotto
78      fib_EntryType    ds.l    1
7C      fib_Size         ds.l    1                ;Lunghezza file
80      fib_NumBlocks    ds.l    1
84      fib_DateStamp    ds.b    ds_SIZEOF        ;Ultima modifica
90      flb_Comment      ds.b    116
        fib_SIZEOF       equ     $104

FIBB_READ       equ     3
FIBF_READ       equ     8
FIBB_WRITE      equ     2
FIBF_WRITE      equ     4

FIBB_EXECUTE    equ     1
FIBF_EXECUTE    equ     2
FIBB_DELETE     equ     0
FIBF_DELETE     equ     1

;InfoData (di un Dischetto)
;-------------------------
;In questa struttura sono contenuti puntatori BCPL,
;per cui dovra' essere giustificata per parola lunga

        CNOP            0,4
00      id_NumSoftErrors ds.l    1
04      id_UnitNumber    ds.l    1
08      id_DiskState     ds.l    1 ;Vedi equ sotto
0C      id_NumBlocks     ds.l    1
10      id_NumBlocksUsed ds.l    1
14      id_BytesPerBlock ds.l    1
18      id_DiskType      ds.l    1
1C      id_VolumeNode    ds.l    1
20      id_InUse         ds.l    1 ;0 se no
        id_SIZEOF        equ     $24

ID_WRITE_PROTECTED      equ     80
ID_VALIDATING           equ     81
ID_VALIDATED            equ     82
```

```
ID_NO_DISK_PRESENT          equ     -1
ID_UNREADABLE_DISK          equ     $42414400       ;'BAD' spostato
ID_NOT_REALLY_DOS           equ     $E444F53        ;'NDOS'
ID_DOS_DISK                 equ     $44F5300        ;'DOS'
ID_KICKSTART_DISK           equ     $B49434B        ;'KICK'


;Error-Codes
;-----------
ERROR_NO_FREE_STORE                 equ     103
ERROR_OBJECT_IN_USE                 equ     202
ERROR_OBJECT_EXISTS                 equ     203
ERR0R_OBJECT_NOT_FOUND              equ     205
ERROR_ACTION_NOT_KNOWN              equ     209
ERROR_INVALID_COMPONENT_NAME        equ     210
ERROR_INVALID_LOCK                  equ     211
ERROR_OBJECT_WRONG_TYPE             equ     212
ERROR_DISK_NOT_VALIDATED            equ     213
ERROR_DISK_WRITE_PROTECTED          equ     214
ERROR_RENAME_ACROSS_DEVICES         equ     215
ERROR_DIRECTORY_NOT_EMPTY           equ     216
ERROR_DEVICE_NOT_MOUNTED            equ     218
ERROR_SEEK_ERROR                    equ     219
ERROR_COMMENT_TOO_BIG               equ     220
ERROR_DISK_FULL                     equ     221
ERROR_DELETE_PROTECTED              equ     222
ERROR_WRITE_PROTECTED               equ     223
ERROR_READ_PROTECTED                equ     224
ERROR_NOT_A_DOS_DISK                equ     225
ERROR_NO_DISK                       equ     226
ERROR_NO_MORE_ENTRIES               equ     232


;Codici di Return consigliati
;----------------------------
RETURN_OK           equ     0       ;Tutto bene
RETURN_WARN         equ     5       ;solo segnalazione/consiglio
RETURN_ERROR        equ     10      ;Errore
RETURN_FAIL         equ     20      ;Errore Totale


;Break-Codes
;-----------
SIGBREAKB_CTRL_C            equ     12
SIGBREAKF_CTRL_C            equ     $1000
SIGBREAKB_CTRL_D            equ     13
SIGBREAKF_CTRL_D            equ     $2000
SIGBREAKB_CTRL_E            equ     14
SIGBREAKF_CTRL_E            equ     $4000
SIGBREAKB_CTRL_F            equ     15
SIGBREAKF_CTRL_F            equ     $8000
```

A4.3 Intuition

```
;Le seguenti Costanti vengono riportate solo come suggerimento.
;Sono state prelevate da altri paragrafi della presente appendice
;al fine di dimostrarne la dipendenza.

bm_SIZEOF       equ     $28     ;da gfx
vp_SIZEOF       equ     $28     ;da view
rp_SIZEOF       equ     $64     ;da rastport
RP_JAM2         equ     1
li_SIZEOF       equ     $66     ; da layers
MN_SIZE         equ     $14     ;da port
TV_SIZE         equ     8       ;da timer


;Fine Aiuto
;-----------------------------------------------------------------


;-----------------------------------
;               Titolo Menu
;-----------------------------------

00      mu_NextMenu     ds.l    1 ;^Prossimo titolo
04      mu_LeftEdge     ds.w    1 ;a sinistra
06      mu_TopEdge      ds.w    1 ;in alto
08      mu_Width        ds.w    1 ;Larghezza
0A      mu_Height       ds.w    1 ;Altezza
0C      mu_Flags        ds.w    1 ;Per i Bit vedi sotto
0E      mu_MenuName     ds.l    1 ;^Testo del titolo
12      mu_FirstItem    ds.l    1 ;^Elenco Item
16      mu_JazzX        ds.w    1 ;interno
18      mu_JazzY        ds.w    1
1A      mu_BeatX        ds.w    1
1C      mu_BeatY        ds.w    1
        mu_SIZEOF       equ     $1E

MENUENABLED     equ     $0001   ;attivo
MIDRAWN         equ     $0100   ;tracciato

;-----------------------------------
;         Item del Menu
;-----------------------------------
00      mi_NextItem             ds.l    1       ;^Prossimo Item
04      mi_LeftEdge             ds.w    1       ;a sinistra
06      mi_TopEdge              ds.w    1       ;in alto
08      mi_Width                ds.w    1       ;Larghezza
0A      mi_Height               ds.w    1       ;Altezza
0C      mi_Flags                ds.w    1       ;Vedi sotto
0E      mi_MutualExclude        ds.l    1       ;1 Bit per Item
12      mi_ItemFIll             ds.l    1       ;^Testo o ^Immagine
16      mi_SelectFIll           ds.l    1       ;vedi sotto
1A      mi_Command              ds.b    1       ;Tasto
```

```
1B      mi_AdjustToWord          ds.b     1
1C      mi_SubItem               ds.l     1        ;^Elenco Sub-item
20      mi_NextSelect            ds.w     1        ;Altro Item?
        mi_SIZEOF                equ      $22

CHECKIT          equu     $0001    ;Item attribuito
ITEHTEXT         equ      $0002    ;Item ha testo o immagine
COMMSEQ          equ      $0004    ;Item ha testo
MENUTOGGLE       equ      $0008
ITEMENABLED      equ      $0010
HIGHFLAGS        equ      $00C0    ;Evidenziazione, quindi:
HIGHIMAGE        equ      $0000    ;Alternativa Image/Text
HIGHCOMP         equ      $0040    ;Complemento di tutti i Bit nell'Item
HIGHBOX          equ      $0080    ;Box intorno alla Item-Box
HIGHNONE         equ      $00C0    ;Nessuna evidenziazione
CHECKED          equ      $0100    ;Check-Mark se selezionato
ISDRAWN          equ      $1000    ;1 se Item su schermo
HIGHITEM         equ      $2000    ;1 se evidenziato
MENUTOGGLED      equ      $4000    ;1 se attivato

NOMENU           equ      $001F    ;1/0= Menu a/da
NOITEM           equ      $003F    ;Item a/da
NOSUB            equ      $001F    ;Subitem a/da
MENUNULL         equ      $FFFF    ;Nessun Item selezionato
CHECKUIDTH       equ      19       ;Spazio per il Check-Mark
COMMWIDTH        equ      27       ;Spazio per il tasto in caso di HighRes
LOWCHECKWIDTH    equ      13       ;in caso di bassa risoluzione


;-----------------------------------
;          Requester
;-----------------------------------

00      rq_OlderRequest  ds.l     1        ;Precedente
04      rq_LeftEdge      ds.w     1        ;a sinistra
06      rq_TopEdge       ds.w     1        ;in alto
08      rq_Width         ds.w     1        ;Larghezza
0A      rq_Height        ds.w     1        ;Altezza
0C      rq_RelLeft       ds.w     1        ;Se puntatore
0E      rq_RelTop        ds.w     1        ;Punto di riferimento
10      rq_ReqGadget     ds.l     1        ;^Elenco Gadget
14      rq_ReqBorder     ds.l     1        ;^Struttura contorno
18      rq_ReqText       ds.l     1        ;^Struttura testo
1C      rq_Flags         ds.w     1        ;Per i Bit vedi sotto
1E      rq_BackFill      ds.b     1        ;Pen per sfondo
1F      rq_AdjustToWord  ds.b     1
20      rq_ReqLayer      ds.l     1        ;^Struttura di Layer
24      rq_ReqPad1       ds.b     32       ;Riservato
44      rq_ReqBMap       ds.l     1        ;^Bit Map Custom
48      rq RWindow       ds.l     1        ;Riservato
4C      rq_ReqPad2       ds.b     36       ;Riservato
        rq_SIZEOF        equ      $70
```

```
POINTREL          equ      $0001    ;1 se relativo al puntatore del Mouse
PREDRAWN          equ      $0002    ;1 se Custom Bit Map
REQOFFWINDOW      equ      $1000    ;se il Req. È fuori finestra
REQACTIVE         equ      $2000    ;0/1 = Req. attivo
SYSREQUEST        equ      $4000    ;solo se Requester di sistema
DEFERREFRESH      equ      $8000    ;


;---------------------------------
;                 Gadgets
;---------------------------------

00      gg_NextGadget     ds.l     1        ;^Prossimo Gadget
04      gg_LeftEdge       ds.w     1        ;a sinistra
06      gg_TopEdge        ds.w     1        ;in alto
08      gg_Width          ds.w     1        ;Larghezza
0A      gg_Height         ds.w     1        ;Altezza
0C      gg_Flags          ds.w     1        ;Per i Bit vedi sotto
0E      gg_Activation     ds.w     1        ;Per i Bit vedi sotto
10      gg_GadgetType     ds.w     1        ;Bool/Str/Prop
12      gg_GadgetRender   ds.l     1        ;^Immagine o ^Contorno
16      gg_SelectRender   ds.l     1        ;^Alternativa di "
1A      gg_GadgetText     ds.l     1        ;^Struttura di testo
1E      gg_MutualExclude  ds.l     1        ;Nessun effetto?
22      gg_SpecialInfo    ds.l     1        ;^Str o PropInfo
26      gg_GadgetID       ds.w     1        ;ID utente a piacere
28      gg_UserData       ds.l     1        ;^a piacere
2C      gg_SIZEOF         equ      $2C

AUTOFRONTPEN      equ      0        ;Valori consigliati per Auto-Request
AUTOBACKPEN       equ      1        ;Vedi IntuitionText
AUTODRAWHODE      equ      1
AUTOLEFTEDGE      equ      6
AUTOTOPEDGE       equ      3
AUTOITEXTFONT     equ      0
AUTONEXTTEXT      equ      0


GADGHIGHB1TS      equ      $0003    ;Nessuna evidenziazione o:
GADGHCOHP         equ      $0000    ;Complemento di tutti i Bits
GADGHBOX          equ      $0001    ;Box e Gadget
GADGHIHAGE        equ      $0002    ;Immagine/Contorno in alternativa
GADGHNONE         equ      $0003    ;Nessuna evidenziazione

GADGIMAGE         equ      $0004    ;0/1=Contorno/Immagine
GRELBOTTOH        equ      $0008    ;0/1= relativo al limite Top/Bottom
GRELRIGHT         equ      $0010    ;0/1- relativo a sinistra/destra
GRELWIDTH         equ      $0020    ;Larghezza Assoluta/Relativa
GRELHEIGHT        equ      $0040    ;Altezza
SELECTED          equ      $0080    ;Selezione da/a
GADGDISABLED      equ      $0100    ;per Gadget a/da
RELVERIFY         equ      $0001    ;Verifica Release
GADGIHMEDIATE     equ      $0002    ;Messaggio immediato se
```

```
ENDGADGET        equ     $0004   ;Requester da schermo
FOLLOWMOUSE      equ     $0008   ;Invio coordinate del Mouse
RIGHTBORDER      equ     $0010   ;Giustificazione a destra
LEFTBORDER       equ     $0020   ;Giustificazione a sinistra
TOPBORDER        equ     $0040   ;Giustificazione in alto
BOTTOMBORDER     equ     $0080   ;Giustificazione in basso
TOGGLESELECT     equ     $0100   ;Gadget selezionato
STRINGCENTER     equ     $0200   ;Giustifica testo
STRINGRIGHT      equ     $0400   ;
LONGINT          equ     $0800   ;Permette Long Int in stringa Gadget
ALTKEYMAP        equ     $1000   ;se disponibile in StringInfo

;Uno di questi tipi deve essere:
BOOLGADGET       equ $0001
GADGET0002       equ $0002
PROPGADGET       equ $0003
STRGADGET        equ $0004

GADGETTYPE       equ $FC00
SYSGADGET        equ $8000       ; Gadget di sistema (passa a  Intuition)
SCRGADGET        equ $4000       ; se sullo schermo
GZZGADGET        equ $2000       ; se nella Window Gimmezerozero
REQGADGET        equ $1000       ; se nel Requester
SIZING           equ $0010       ; tipi di Sys
WDRAGGING        equ $0020
SDRAGGING        equ $0030
WUPFRONT         equ $0040
SUPFRONT         equ $0050
WDOWNBACK        equ $0060
SDOWNBACK        equ $0070
CLOSE            equ $0080


;PropInfo (per Proportional-Gadgets)
;-------------------------------------
00      pi_Flags        ds.w    1       ;vedi sotto
02      pi_HorizPot     ds.w    1       ;Orizontale %
04      pi_VertPot      ds.w    1       ;Verticale %
06      pi_HorizBody    ds.w    1       ;Visualizzarne
08      pi_VertBody     ds.w    1       ;uno dei due
0A      pi_CWidth       ds.w    1       ;Larghezza
0C      pi_CHeight      ds.w    1       ;Altezza
0E      pi_HPotRes      ds.w    1       ;Scritta in Orizzontale
10      pi_VPotRes      ds.w    1       ;Scritta in Verticale
12      pi_LeftBorder   ds.w    1       ;Posizione bordo a sinistra
14      pi_TopBorder    ds.w    1       ;Posizione bordo in alto
16      pi_SIZEOF       equ     $16


AUTOKNOB         equ     $0001   ;Pulsante Autom.
FREEHORIZ        equ     $0002   ;Movimento orizontale pulsante
FREEVERT         equ     $0004   ;Movimento verticale pulsante
PROPBORDERLESS   equ     $0008   ;Senza bordo
```

```
KNOBHIT          equ     $0100   ;1 se il pulsante viene premuto
KNOBHMIN         equ     6       ;Limiti :
KNOBVMIN         equ     4
MAXBODY          equ     $FFFF
MAXPOT           equ     $FFFF

; StringInfo (per String-Gadgets)
;-----------------------------------

00      si_Buffer        ds.l    1        ;^Buffer di lavoro
04      si_UndoBuffer    ds.l    1        ;^Undo-Buffer oppure 0
08      si_BufferPos     ds.w    1        ;Posizione iniziale cursore
0A      si MaxChars      ds.w    1        ;Dimensione Buffer + 1;
0C      si_DispPos       ds.w    1        ;Posizione carattere cursore
0E      si_UndoPos       ds.w    1        ;Cursore nell'Undo-Buffer
10      si_NumChars      ds.w    1        ;Carattere nel Buffer
12      si_DispCount     ds.w    1        ;Carattere visibile
14      si_CLeft         ds.w    1        ;Posizione bordo
16      si_CTop          ds.w    1        ;
18      si LayerPtr      ds.l    1        ;^Layer del Gadget
1C      si_LongInt       ds.l    1        ;Long Int qui
20      si_AltKeyNap     ds.l    1        ;^Keymap prorpio
        si_SIZEOF        equ     $24

;---------------------------------------
;            Intuition Text
;---------------------------------------

00      it_FrontPen      ds.b    1        ;Colore primo piano
01      it_BackPen       ds.b    1        ;Colore sfondo
02      it_DrawMode      ds.b    1        ;JAH1, JAM2 oppure XOR
03      it_AdjustToWord  ds.b    1        ;
04      it_LeftEdge      ds.w    1        ;Posizione a sinistra
06      it_TopEdge       ds.w    1        ;Posizione in alto
08      it_ITextFont     ds.l    1        ;^Struttura Font oppure 0
0C      it_IText         ds.l    1        ;^Stringa di testo (0-term.)
10      it_NextText      ds.l    1        ;^Prossima Struttura  o 0
        it_SIZEOF        equ     $14


;---------------------------------------
;            Borders (Poligoni)
;---------------------------------------

00      bd_LeftEdge      ds.w    1        ;Inizio a sinistra
02      bd_TopEdge       ds.w    1        ;Inizio in alto
04      bd_FrontPen      ds.b    1        ;Colore primo piano
05      bd_BackPen       ds.b    1        ;senza effetto
06      bd_DrawHode      ds.b    1        ;JAM1 oppure XOR
07      bd_Count         ds.b    1        ;Numero Coppie
08      bd_XY            ds.l    1        ;^Matrice con coppia
0C      bd_NextBorder    ds.l    1        ;^Prossimo oppure 0
10      bd SIZEOF        equ     $10
```

```
;-------------------------------------
;                Images
;-------------------------------------

00      ig_LeftEdge      ds.w     1        ;Posizione a sinistra
02      ig_TopEdge       ds.w     1        ;Posizione in alto
04      ig_Width         ds.w     1        ;Larghezza
06      ig_Height        ds.w     1        ;Altezza
08      ig_Depth         ds.w     1        ;Numero di Bitplanes
0A      ig_ImageData     ds.l     1        ;^Campione di Bit
0E      ig_PlanePick     ds.b     1        ;Plane utilizzati
0F      ig_PlaneOnOff    ds.b     1        ;
10      ig_NextImage     ds.l     1        ;^Prossima struttura
14      ig_SIZEOF        equ      $14



;-------------------------------------
;           Intuition Message
;-------------------------------------

00      im_ExecMessage   ds.b     MN_SIZE          ;riservato
14      im_Class         ds.l     1                ;Bit come Flag IDCMP
18      im_Code          ds.w     1                ;Valori qui
1A      im_Qualifier     ds.w     1                ;per  RAW-IO
1C      im_IAddress      ds.l     1                ;Indirizzo degli oggetti
20      im_MouseX        ds.w     1                ;Coordinate Mouse
22      lm_MouseY        ds.w     1                ;
24      im_Seconds       ds.l     1                ;Ora del sistema
28      im_Micros        ds.l     1                ;
2C      im_IDCMPWindow   ds.l     1                ;Indirizzo finestra
30      im_SpecialLink   ds.l     1                ;riservato
        im_SIZEOF        equ      $34

SIZEVERIFY       equ      $00000001        ;Messaggio se tentativo di Sizing
NEWSIZE          equ      $00000002        ;Messaggio se Sizing completato
REFRESHWINDOW    equ      $00000004        ;Messaggio se Refresh necessario
MOUSEBUTTONS     equ      $00000008        ;Messaggio se Event da Mouse
MOUSEMOVE        equ      $00000010        ;
GADGETDOWN       equ      $00000020        ;Messaggio se Event da Gadget
GADGETUP         equ      $00000040        ;
REQSET           equ      $00000080        ;Messaggio se Requester
MENUPICK         equ      $00000100        ;Messaggio se Event da Menu
CLOSEWINDOW      equ      $00000200        ;Messaggio se Close_Gadget
RAWKEY           equ      $00000400        ;Messaggio se Raw-Key
REQVERIFY        equ      $00000800        ;Attesa permesso dal Requester
REQCLEAR         equ      $00001000        ;Mess. se ultimo Req. eliminato
MENUVERIFY       equ      $00002000        ;Attesa visualizzazione Menu
NEWPREFS         equ      $00004000        ;Messaggio se modifica Prefs.
DISKINSERTED     equ      $00008000        ;Messaggio se Dischetto
DISKREMOVED      equ      $00010000        ;inserito/estratto
WBENCHMESSAGE    equ      $00020000        ;
ACTIVEWINDOW     equ      $00040000        ;
INACTIVEWINDOW   equ      $00080000        ;
```

```
DELTAMOVE        equ     $00100000        ;relativo al posizione del Mouse
VANILLAKEY       equ     $00200000        ;Messaggio se Key in codice
INTUITICKS       equ     $00400000        ;Messaggio dopo 1/10 secondo
LONELYMESSAGE    equ     $80000000

;per Menu-Verify:
MENUHOT          equ     $0001    ;Cancel deve venire verificato
MENUCANCEL       equ     $0002    ;Hot Reply cancella il Menu
MENUWAITING      equ     $0003    ;Int. Aspetta risposta Reply

WBENCHOPEN       equ     $0001
WBENCHCLOSE      equ     $0002

;----------------------------------------
;                NewWindow
;----------------------------------------
00      nw_LeftEdge      ds.w 1           ;Sinistra
02      nw_TopEdge       ds.w 1           ;in alto
04      nw_Width         ds.w 1           ;Larghezza
06      nw_Height        ds.w 1           ;Altezza
08      nw_DetailPen     ds.b 1           ;Penna fine
09      nw_BlockPen      ds.b 1           ;Penna grossa
0A      nw_IDCMPFlags    ds.l 1           ;Per i Bit vedi sotto
0E      nw_Flags         ds.l 1           ;
12      nw_FirstGadget   ds.l 1           ;^Gadget utente
16      nw_CheckMark     ds.l 1           ;^Checkmark utente
1A      nw_Title         ds.l 1           ;^Testo titolo
1E      nw_Screen        ds.l 1           ;^Screen
22      nw_BitMap        ds.l 1           ;^Bitmap utente
26      nw_MinWidth      ds.w 1           ;Larghezza minima
28      nw_MinHeight     ds.w 1           ;Altezza minima
2A      nw_MaxWidth      ds.w 1           ;Max. Larghezza
2C      nw_MaxHeight     ds.w 1           ;Altezza massima
2E      nw_Type          ds.w 1           ;Tipo di Screen
        nw SIZE          equ $30



WINDOWSIZING     equ     $0001    ;Gadget permessi
WINDOWDRAG       equ     $0002
WINDOWDEPTH      equ     $0004
WINDOWCLOSE      equ     $0008

SIZEBRIGHT       equ     $0010    ;Dim. Gadget destro esterno
SIZEBBOTTOM      equ     $0020    ;Dim. Gadget destro interno

REFRESHBITS      equ     $00C0    ;Uno di questi:
SMART_REFRESH    equ     $0000    ;
SIMPLE_REFRESH   equ     $0040    ;
SUPER_BITMAP     equ     $0080    ;
OTHER_REFRESH    equ     $00C0    ;

BACKDROP         equ     $0100    ;Se Backdrop-Window
REPORTMOUSE      equ     $0200    ;Messaggio se Event da Mouse
```

```
GIMMEZEROZERO     equ     $0400               ;Se richiesto
BORDERLESS        equ     $0800               ;Se senza bordo

ACTIVATE          equ     $1000               ;Attivo dopo Open
WINDOWACTIVE      equ     $2000               ;Messaggio se attivato
INREQUEST         equ     $4000               ;Wd in modo Request
MENUSTATE         equ     $8000               ;Messaggio se Menu

RMETRAP           equ     $00010000           ;Messaggio se tasto destro Mouse
NOCAREREFRESH     equ     $00020000           ;Nessun messaggio con Refresh

WINDOWREFRESH     equ     $01000000
WBENCHWINDOW      equ     $02000000
WINDOWTICKED      equ     $04000000

SUPER_UNUSED      equ     $FCFC0000


;--------------------------------------
;                 Window
;--------------------------------------

00      wd_NextWindow     ds.l    1           ;^Prossima Window
04      wd_LeftEdge       ds.w    1           ;a sinistra
06      wd_TopEdge        ds.w    1           ;in alto
08      wd_Width          ds.w    1           ;Larghezza
0A      wd_Height         ds.w    1           ;Altezza
0C      wd_MouseY         ds.w    1           ;Mouse Y
0E      wd_MouseX         ds.w    1           ;Mouse X

10      wd_MinWidth       ds.w    1           ;Larghezza Min.
12      wd_MinHeight      ds.w    1           ;Altezza Min.
14      wd_MaxWidth       ds.w    1           ;Larghezza Max.
16      wd_MaxHeight      ds.w    1           ;Altezza Max.
18      wd_Flags          ds.l    1           ;Bit vedi sopra
1C      wd_HenuStrip      ds.l    1           ;^Elenco Menu
20      wd_Title          ds.l    1           ;^Testo titolo
24      wd_FirstRequest   ds.l    1           ;^Primo Request
28      wd_DMRequest      ds.l    1           ;^DM Request
2C      wd_ReqCount       ds.w    1           ;Numero Request
2E      wd_WScreen        ds.l    1           ;^Screen
32      wd_RPort          ds.l    1           ;^RastPort
36      wd_BorderLeft     ds.b    1           ;Posizione attuale di:
37      wd_BorderTop      ds.b    1           ;
38      wd_BorderRight    ds.b    1           ;
39      wd_BorderBottom   ds.b    1           ;
3A      wd_BorderRPort    ds.l    1           ;^RastPortG00-Wd esterno
3E      wd_FirstGadget    ds.l    1           ;^Elenco Gadget
42      wd_Parent         ds.l    1           ;in elenco
46      wd_Descendant     ds.l    1           ;
4A      wd_Pointer        ds.l    1           ;^Struttura Mouse
4E      wd_PtrHeight      ds.b    1           ;Altezza
4F      wd_PtrWidth       ds.b    1           ;Larghezza
```

```
50      wd_XOffset       ds.b    1        ;Offset
51      wd_YOffset       ds.b    1        ;
52      wd_IDCHPFlags    ds.l    1        ;per i Bit vedi in alto
56      wd_UserPort      ds.l    1        ;^Porta di ricezione
5A      wd_WindowPort    ds.l    1        ;^Porta di invio
5E      wd_MessageKey    ds.l    1        ;^Messaggio Int.
62      wd_DetailPen     ds.b    1        ;Penna sottile
63      wd_BlockPen      ds.b    1        ;Penna grossa
64      wd_CheckMark     ds.l    1        ;^Checkmark utente
68      wd_ScreenTitle   ds.l    1        ;^Titolo Screen (0)
6C      wd_GZZMouseX     ds.w    1        ;Solo se G00-Window
6E      wd_GZZMouseY     ds.w    1        ;
70      wd_GZZWidth      ds.w    1        ;
72      wd_GZZHeight     ds.w    1        ;
74      wd_ExtData       ds.l    1        ;Due puntatori per
78      wd_UserData      ds.l    1        ;User
7C      wd_WLayer        ds.l    1        ;^Layer del Wd


;-----------------------------------
;                NewScreen
;-----------------------------------

00      ns_LeftEdge      ds.w    1        ; a sinistra
02      ns_TopEdge       ds.w    1        ;in alto
04      ns_Width         ds.w    1        ;Larghezza
06      ns_Helght        ds.w    1        ;Altezza
08      ns_Depth         ds.w    1        ;Bitplanes
0A      ns_DetailPen     ds.b    1        ;Penna fine
0B      ns_BlockPen      ds.b    1        ;Penna grossa
0C      ns_ViewModes     ds.w    1        ;0, HIRES usw.
0E      ns_Type          ds.w    1        ;CUSTOM o Bitmap
10      ns_Font          ds.l    1        ;^Font o 0
14      ns_DefaultTitle  ds.l    1        ;^Testo titolo
18      ns_Gadgets       ds.l    1        ;Imposta sempre 0!
1C      ns_CustomBitMap  ds.l    1        ;^sulla tua bitmap
        ns_SIZEOF        equ     $20

SCREENTYPE       equ     $000F ;tutti i tipi
WBENCHSCREEN     equ     $0001 ;Workbench
CUSTOMSCREEN     equ     $000F ;proprio
SHOWTITLE        equ     $0010 1 ;se chiamata di ShowTitle
BEEPING          equ     $0020 1 ;quando Beep (Blink)
CUSTOMBITMAP     equ     $0040 1 ;se propria bitmap

FILENAME_SIZE    equ     30

POINTERSIZE      equ     36

TOPAZ_EIGHTY     equ     8
TOPAZ_SIXTY      equ     9
```

```
;-----------------------------------
;                Screen
;-----------------------------------

000     sc_NextScreen    ds.l    1        ;^Prossimo
004     sc_FirstWindow   ds.l    1        ;^Prima Window
008     sc_LeftEdge      ds.w    1        ;a sinistra
00A     sc_TopEdge       ds.w    1        ;in alto
00C     sc_Width         ds.w    1        ;Larghezza
00E     sc_Height        ds.w    1        ;Altezza
010     sc_MouseY        ds.w    1        ;posizione del mouse
012     sc_HouseX        ds.w    1        ;posizione del mouse
014     sc_Flags         ds.w    1        ;Per i Bits vedi in alto
016     sc_Title         ds.l    1        ;^Testo titolo
01A     sc_DefaultTitle  ds.l    1        ;^Testo per Wd ohne
01E     sc_BarHeight     ds.b    1        ;Altezza della Barra
01F     sc_BarVBorder    ds.b    1        ;per Screen
020     sc_BarHBorder    ds.b    1        ;e tutte le sue Window
021     sc_MenuVBorder   ds.b    1
022     sc_MenuHBorder   ds.b    1
023     sc_WBorTop       ds.b    1
024     sc_WBorLeft      ds.b    1
025     sc_WBorRight     ds.b    1
026     sc_WBorBottom    ds.b    1
027     sc_AdjustToWord  ds.b    1
028     sc_Font          ds.l    1        ;^Font di default
02C     sc_ViewPort      ds.b    $64      ;Tipo di visualizzazione
054     sc_RastPort      ds.b    rp_SIZEOF ;Tipo di carattere
0B8     sc_BitMap        ds.b    $28      ;Struttura Bitmap esterna
0E0     sc_LayerInfo     ds.b    li_SIZEOF ;Layer-Info
146     sc_FirstGadget   ds.l    1        ;^Elenco Gadget
14A     sc_DetailPen     ds.b    1        ;Pen tracciato
14B     sc_BlockPen      ds.b    1
14C     sc_SaveColor0    ds.w    1        ;Per Beep
14E     BarLayer         ds.l    1        ;Omettere sc_
152     sc_ExtData       ds.l    1        ;^Dati utente
156     sc_UserData      ds.l    1        ;idem
        sc_SIZEOF        equ     $15A


;-----------------------------------------------
;       Preferences (vedi anche Equates sotto)
;-----------------------------------------------

00      pf_FontHeight            ds.b    1
01      pf_PrinterPort           ds.b    1
02      pf_BaudRate              ds.w    1
04      pf_KeyRptSpeed           ds.b    TV_SIZE
0C      pf_KeyRptDelay           ds.b    TV_SIZE
14      pf_DoubleClick           ds.b    TV_SIZE
1C      pf_PointerHatrix         ds.b    POINTERSIZE*2
64      pf_XOffset               ds.b    1
65      pf_YOffset               ds.b    1
```

```
66      pf_color17              ds.w    1
68      pf_color18              ds.w    1
6A      pf_color19              ds.w    1
6C      pf_PointerTicks         ds.w    1
6E      pf_color0               ds.w    1
70      pf_color1               ds.w    1
72      pf_color2               ds.w    1
74      pf_color3               ds.w    1
76      pf_ViewXOffset          ds.b    1
77      pf_ViewYOffset          ds.b    1
78      pf_ViewInitX            ds.w    1
7A      pf_ViewInitY            ds.w    1
7C      EnableCLI               ds.w    1
7E      pf_PrinterType          ds.w    1
80      pf_Printerfilename      ds.b    FILENAME_SIZE
9E      pf_PrintPitch           ds.w    1
A0      pf_PrintQuality         ds.w    1
A2      pf_PrintSpacing         ds.w    1
A4      pf_PrintLeftMargin      ds.w    1
A6      pf_PrintRightHargin     ds.w    1
A8      pf_PrintImage           ds.w    1
AA      pf_PrintAspect          ds.w    1
AC      pf_PrintShade           ds.w    1
AE      pf_PrintThreshold       ds.w    1
B0      pf_PaperSlze            ds.w    1
B2      pf_PaperLength          ds.w    1
B4      pf_PaperType            ds.w    1
B6      pf_padding              ds.b    50
        pf_SIZEOF               equ     $E8


PARALLEL_PRINTER equ    $00
SERIAL_PRINTER   equ    $01
BAUD_110         equ    $00
BAUD_300         equ    $01
BAUD_1200        equ    $02
BAUD_2400        equ    $03
BAUD_4800        equ    $04
BAUD_9600        equ    $05
BAUD_19200       equ    $06
BAUD_MIDI        equ    $07

FANFOLD          equ    $00
SINGLE           equ    $80
PICA             equ    $000
ELITE            equ    $400

FINE             equ    $800
DRAFT            equ    $000
LETTER           equ    $100

SIX_LPI          equ    $000
```

```
EIGHT_LPI        equ      $200

IMAGE_POSITIVE   equ      0
IMAGE_NEGATIVE   equ      1

ASPECT_HORIZ     equ      0
ASPECT_VERT      equ      1

SHADE_BW         equ      $00
SHADE_GREYSCALE  equ      $01
SHADE_COLOR      equ      $02

US_LETTER        equ      $00
US_LEGAL         equ      $10
N_TRACTOR        equ      $20
W_TRACTOR        equ      $50
CUSTOM           equ      $40
CUSTOM_NAME      equ      $00

ALPHA_P_101      equ      $01
BROTHER_15XL     equ      $02
CBM_MPS1000      equ      $03
DIAB_650         equ      $04
DIAB_ADV_D25     equ      $05
DIAB_C_150       equ      $06
EPSON            equ      $07
EPSON_JX_80      equ      $08
OKIMATE_20       equ      $09
QUME_LP_20       equ      $0A
HP_LASERJET      equ      $0B
HP_LASERJET_PLUS equ      $0C

;-------------------------------------
;                Remember
;-------------------------------------
00       rm_NextRemember   ds.l     1        ;^Prossimo nodo
04       rm_RememberSize   ds.l     1        ;Dimensioni
08       rm_Memory         ds.l     1        ;^Indirizzo
0C       rm_SIZEOF         ds.w     0

;---------------------------------
;               Alerts
;---------------------------------

ALERT_TYPE       equ      $80000000
RECOVERY_ALERT   equ      $00000000
DEADEND_ALERT    equ      $80000000
```

A4.4 Graphics

```
;Importato da :
;========================================

MP_SIZE          equ      $22      ;ports
LH_SIZE          equ      $0E      ;lists
MN_SIZE          equ      $14      ;ports
IS_SIZE          equ      $16      ;libraries
LIB_SIZE         equ      $22


;Layer-Structure
;---------------

00       lr_Front                  ds.l 1   ; ^Layer sopra questo
04       lr_Back                   ds.l 1   ;^Layer sotto questo
08       lr_ClipRect               ds.l 1   ;^Clipping struttura rettangolare
0C       lr_RastPort               ds.l 1   ;^Rastport
10       lr_MinX                   ds.w 1   ;Clipping rettangolare:
12       lr_MinY                   ds.w 1
14       lr_MaxX                   ds.w 1
16       lr_MaxY                   ds.w 1
18       lr_Lock                   ds.b 1   ;Blocco Task del Layer
19       lr_LockCount              ds.b 1   ;Numero dei Task
1A       lr_LayerLockCount         ds.b 1   ;Per questo Layer
1B       lr_reserved               ds.b 1
1C       lr_reserved1              ds.w 1
1E       lr_Flags                  ds.w 1   ;16 Bit = Tipo
20       lr_SuperBitHap            ds.l 1   ;^Super-Bitmap
24       lr_SuperClipRect          ds.l 1   ;^ClipRect se S-Bitmap-Lay.
28       lr_Window                 ds.l 1   ;^Intuition-Window
2C       lr_Scroll_X               ds.w 1   ;Larghezza scroll in Pixels
2E       lr_Scroll_Y               ds.w 1
30       lr_LockPort               ds.b MP_SIZE ;Nome Msg-Port
52       lr_LockMessage            ds.b MN_SIZE ;Struttura Msg
66       lr_ReplyPort              ds.b MP_SIZE ;Nome Msg-Port
88       lr_l_LockMessage          ds.b MN_SIZE ;Struttura Msg
9C       lr_DamageList             ds.l 1 ;^Struttura regione
A0       lr_cliprects              ds.l 1 ;^Clip-Rect
A4       lr_LayerInfo              ds.l 1 ;^Struttura Layerinfo
A8       lr_LayerLocker            ds.l 1 ;^Struttura Task
AC       lr_SuperSaverClipRects    ds.l 1 ; Uso Sistema:
B0       lr_cr                     ds.l 1
B4       lr_cr2                    ds.l 1
B8       lr_crnew                  ds.l 1
BC       lr_p1                     ds.l 1
C0       lr_SIZEOF                 ds.w 0


; Clip-Rect
;------------------------------------

00       cr_Next          ds.l     1        ;^Successivo
04       cr_Prev          ds.l     1        ;^Precedente
```

```
08      cr_LOBS         ds.l    1       ; Uso Sistema
0C      cr_BitMap       ds.l    1       ;^Super-Bitmap
10      cr_MinX         ds.w    1       ;Rettangolo:
12      cr_MinY         ds.w    1
14      cr_MaxX         ds.w    1
16      cr_MaxY         ds.w    1
18      cr_p1           ds.l    1       ;Uso Sistema
1C      cr_p2           ds.l    1
20      cr_reserved     ds.l    1
24      cr_Flags        ds.l    1
28      cr_SIZEOF       ds.w    0



ISLESSX         equ     1
ISLESSY         equ     2
ISGRTRX         equ     4
ISGRTRY         equ     8


; copper
;--------

COPPER_MOVE     equ     0       ;Pseudo-Op-Codes
COPPER_WAIT     equ     1
CPRNXTBUF       equ     2
CPR_NT_LOF      equ     $8000
CPR_NT_SHT      equ     $4000

00      ci_OpCode       ds.w    1       ;Op-Codes
02      ci_nxtlist      ds.b    0
02      ci_VWaitPos     ds.b    0
02      ci_DestAddr     ds.b    2
04      ci_HWaitPos     ds.b    0
04      ci_DestData     ds.b    2
06      ci_SIZEOF       ds.w    0


00      crl_Next        ds.l    1 ;^Elenco attuale dei Copper
04      crl_start       ds.l    1
08      crl_MaxCount    ds.w    1
0A      crl_SIZEOF      ds.w    0

; Elenco dei Copper
;------------------
00      cl_Next         ds.l    1 ;^Successivo
04      cl_CopList      ds.l    1 ; Uso Sistema
08      cl_ViewPort     ds.l    1
0C      cl_CopIns       ds.l    1
10      cl_CopPtr       ds.l    1
14      cl_CopLStart    ds.l    1
18      cl_CopSStart    ds.l    1
1C      cl_Count        ds.w    1
1E      cl_MaxCount     ds.w    1
20      cl_DyOffset     ds.w    1
```

```
22      cl_SIZEOF                 ds.w     0

00      ucl_Next                  ds.l     1    ;Cop-List-Header
04      ucl_FirstCopList          ds.l     1
08      ucl_CopList               ds.l     1
0C      ucl_SIZEOF                ds.w     0

00      copinit_diagstrt          ds.b     8    ;Struttura interna di Cop
08      copinit_sprstrtup         ds.b     80
58      copinit_sprstop           ds.b     4
5C      copinit_SIZEOF            ds.w     0


;Elementi grafici
;----------------

SUSERFLAGS       equ      $0F
VSB_VSPRITE      equ      0
VSF_VSPRITE      equ      1
VSB_SAVEBACK     equ      1
VSF_SAVEBACK     equ      2
VSB OVERLAY      equ      2
VSF_OVERLAY      equ      4
VSB_MUSTDRAW     equ      3
VSF_MUSTDRAW     equ      8
VSB_BACKSAVED    equ      8
VSF_BACKSAVED    equ      $100
VSB_BOBUPDATE    equ      9
VSF_BOBUPDATE    equ      $200
VSB_GELGONE      equ      10
VSF_GELGONE      equ      $400
VSB_VSOVERFLOW   equ      11
VSF_VSOVERFLOW   equ      $800

BUSERFLAGS       equ      $0FF
BB_SAVEBOB       equ      0
BF_SAVEBOB       equ      1
BB_BOBISCOMP     equ      1
BF_BOBISCOHP     equ      2
BB_BWAITING      equ      8
BF_BWAITING      equ      $100
BB_BDRAWN        equ      9
BF_BDRAWN        equ      $200
BB_BOBSAWAY      equ      10
BF_BOBSAWAY      equ      $400
BB_B0BNIX        equ      11
BF_BOBNIX        equ      $800
BB_SAVEPRESERVE  equ      12
BF_SAVEPRESERVE  equ      $1000
BB_OUTSTEP       equ      13
BF_OUTSTEP       equ      $2000

ANFRACSIZE       equ      6
```

```
ANIMHALF         equ      $20
RINGTRIGGER      equ      1

;Struttura V degli Sprite (anche per i Bob)
;------------------------------------------

00      vs_NextVSprite   ds.l     1        ;^Successivo
04      vs_PrevVSprite   ds.l     1        ;^Precedente
08      vs_DrawPath      ds.l     1        ;Uso sistema
0C      vs_ClearPath     ds.l     1        ;Uso sistema
10      vs_Oldy          ds.w     1        ;Pos. Y precedente
12      vs_Oldx          ds.w     1        ;Pos. X precedente
14      vs_VSFlags       ds.w     1        ;
16      vs_Y             ds.w     1
18      vs_X             ds.w     1
1A      vs_Height        ds.w     1        ;Altezza dello Sprite
1C      vs_Width         ds.w     1        ;Larghezza
1E      vs_Depth         ds.w     1        ;Bit-Plane
20      vs_MeMask        ds.w     1        ;Maschere per
22      vs_HitMask       ds.w     1        ;Gestione collisioni
24      vs_ImageData     ds.l     1        ;^Dati
28      vs_BorderLine    ds.l     1        ;^Buffer
2C      vs_CollHask      ds.l     1        ;^Maschera collisione
30      vs_SprColors     ds.l     1        ;^Tabella colori
34      vs_VSBob         ds.l     1        ;^Bob se Bob
38      vs_PlanePick     ds.b     1        ;Maschera Plane se Bob
39      vs_PlaneOnOff    ds.b     1
3A      vs_SUserExt      ds.w     0        ;eventuali estensioni utente
3A      vs_SIZEOF        ds.w     0


;Bobs
;-----

00      bob_BobFlags     ds.w     1        ;Bit dell’aspetto
02      bob_SaveBuffer   ds.l     1        ;^Buffer
06      bob_ImageShadou  ds.l     1        ;^Maschera ombra
0A      bob_Before       ds.l     1        ;^Precedente
0E      bob_After        ds.l     1        ;^Successivo
12      bob_BobVSprite   ds.l     1
16      bob_BobComp      ds.l     1
1A      bob_DBuffer      ds.l     1
1E      bob_BUserExt     ds.w     0
1E      bob_SIZEOF       ds.w     0

;Svolgimento dell’animazione:
;----------------------------
00      ac_CompFlags     ds.w     1 ;Bit del tipo
02      ac_Timer         ds.w     1 ;Tempo reale
04      ac_TimeSet       ds.w     1 ;Impostazione
06      ac_NextComp      ds.l     1 ;^Successivo
0A      ac_PrevComp      ds.l     1 ; ^Precedente
0E      ac_NextSeq       ds.l     1 ;dto in
12      ac_PrevSeq       ds.l     1 ;sequenza dei caratteri
16      ac_AnimCRoutine  ds.l     1 ;^Routine di Exit (0)
```

```
1A      ac_YTrans        ds.w    1       ;Distanza iniziale
1C      ac_XTrans        ds.w    1
1E      ac_HeadOb        ds.l    1       ;^Struttura AminOb-Structure
22      ac_AnimBob       ds.l    1       ; ^Bob-Structure
26      ac_SIZE          ds.w    0


; Oggetto dell'animazione
;-----------------------
00      ao_NextOb        ds.l    1       ;^Successivo
04      ao_PrevOb        ds.l    1       ;^Precedente
08      ao_Clock         ds.l    1       ;Chiamata
0C      ao_AnOldY        ds.w    1       ;Vecchio stato
0E      ao_AnOldX        ds.w    1
10      ao_AnY           ds.w    1       ;è
12      ao_AnX           ds.w    1
14      ao_YVel          ds.w    1       ;Velocità
16      ao_XVel          ds.w    1
18      ao_XAccel        ds.w    1       ;Accelerazione
1A      ao_YAccel        ds.w    1
1C      ao_RingYTrans    ds.w    1       ;Incrementi
1E      ao_RingXTrans    ds.w    1
20      ao_AnimORoutine  ds.l    1       ;^Routine
24      ao_HeadComp      ds.l    1       ;^Primo oggetto
28      ao_AUserExt      ds.w    0
28      ao_SIZEOF        ds.w    0

00      dbp_BufY         ds.w    1       ;Buffer intermedio
02      dbp_BufX         ds.w    1
04      dbp_BufPath      ds.l    1
08      dbp_BufBuffer    ds.l    1
0C      dbp_BufPlanes    ds.l    1
10      dbp_SIZEOF       ds.w    0


;gfxbase
;-------

22      gb_ActiView      ds.l    1
26      gb_copinit       ds.l    1
2A      gb_cia           ds.l    1
2E      gb_blitter       ds.l    1
32      gb_LOFlist       ds.l    1
36      gb_SHFlist       ds.l    1
3A      gb_blthd         ds.l    1
3E      gb_blttl         ds.l    1
42      gb_bsblthd       ds.l    1
46      gb_bsblttl       ds.l    1
4A      gb_vbsrv         ds.b    IS_SIZE
60      gb_timsrv        ds.b    IS_SIZE
76      gb_bltsrv        ds.b    IS_SIZE
8C      gb_TextFonts     ds.b    LH_SIZE
9A      gb_DefaultFont   ds.l    1
9E      gb_Modes         ds.w    1
```

```
A0      gb_VBlank               ds.b    1
A1      gb_Debug                ds.b    1
A2      gb_BeamSync             ds.w    1
A4      gb_system_bplcon0       ds.w    1
A6      gb_SpriteReserved       ds.b    1
A7      gb_bytereserved         ds.b    1
A8      gb_Flags                ds.w    1
AA      gb_BlitLock             ds.w    1
AC      gb_BlitNest             ds.w    1
AE      gb_BlitWaitQ            ds.b    LH_SIZE
BC      gb_BlitOuner            ds.l    1
C0      gb_TOF_WaitQ            ds.b    LH_SIZE
CE      gb_DisplayFlags         ds.w    1
D0      gb_SimpleSprites        ds.l    1
D4      gb_MaxDisplayRov        ds.w    1
D6      gb_reserved             ds.b    1
DE      gb_SIZE                 ds.b    0


OWNBLITTERn     equ     0
QBOWNERn        equ     1
QBOWNER         equ     2



;GFX
;--------------

BITSET  equ     $8000
BITCLR  equ     0
AGNUS   equ     1
DENISE  equ     1

00      bm_BytesPerRov  ds.w    1
02      bm_Rows         ds.w    1
04      bm_Flags        ds.b    1
05      bm_Depth        ds.b    1
06      bm_Pad          ds.w    1
08      bm_Planes       ds.b    32
28      bm_SIZEOF       ds.w    0

00      ra_MinX         ds.w    1
02      ra_MinY         ds.w    1
04      ra_MaxX         ds.w    1
06      ra_MaxY         ds.w    1
08      ra_SIZEOF       ds.w    0

;layers
;------

00      lie_env                 ds.b    52
34      lie_mem                 ds.b    LH_SIZE
42      lie_FreeClipRects       ds.l    1
46      lie_blitbuff            ds.l    1
```

```
4A      lie_SIZEOF                ds.w     0

LMN_REGION       equ      -1


;memory
;------
memnode_succ     ds.l     1
memnode_pred     ds.l     1
memnode_where    ds.l     1
memnode_hov_big  ds.l     1
memnode_SIZEOF   ds.w     0


;Struttura di LayerInfo
;----------------------
00      li_top_layer              ds.l     1 ;^Layerin alto
04      li_check_lp               ds.l     1 ;Uso sistema:
08      li_obs                    ds.l     1
0C      li_RP_ReplyPort           ds.b     MP_SIZE
2E      li_LockPort               ds.b     MP_SIZE
50      li_Lock                   ds.b     1
51      li_broadcast              ds.b     1
52      li_locknest               ds.b     1
53      li_pad                    ds.b     1
54      li_Locker                 ds.l     1
58      li_bytereserved           ds.b     2
5A      li_wordreserved           ds.b     4
5E      li_longreserved           ds.b     4
62      li_LayerInfo_extra        ds.l     1
66      li_SIZEOF                 ds.w     0

NEWLAYERINFO_CALLED       equ      1


;rastport
;--------


00      tr_RasPtr                 ds.l     1
04      tr_Size                   ds.l     1
08      tr_SIZEOF                 ds.w     0

00      gi_sprRsrvd               ds.b     1
01      gi_Flags                  ds.b     1
02      gi_gelHead                ds.l     1
06      gi_gelTail                ds.l     1
0A      gi_nextLine               ds.l     1
0E      gi_lastColor              ds.l     1
12      gi_collHandler            ds.l     1
16      gi_leftmost               ds.w     1
18      gi_rightmost              ds.w     1
1A      gi_topmost                ds.w     1
1C      gi_bottommost             ds.w     1
1E      gi_firstBlissObj          ds.l     1
22      gi_lastBlissObj           ds.l     1
```

```
26      gi_SIZEOF                ds.w    0

RPB_FRST_DOT      equ      0
RPF_FRST_DOT      equ      1
RPB_ONE_DOT       equ      1
RPF_ONE_DOT       equ      2
RPB_DBUFFER       equ      2
RPF_DBUFFER       equ      4
RPB_AREAOUTLINE   equ      3
RPF_AREAOUTLINE   equ      8
RPB_NOCROSSFILL   equ      5
RPF_NOCROSSFILL   equ      32

RP_JAM1           equ      0
RP_JAM2           equ      1
RP_COMPLEMENT     equ      2
RP_INVERSVID      equ      4

RPB_TXSCALE       equ      0
RPF_TXSCALE       equ      1


;Struttura di RastPort
;---------------------
00      rp_Layer          ds.l     1        ;^Layer
04      rp_BitMap         ds.l     1        ;^Bitmap
08      rp_AreaPtrn       ds.l     1        ;^Riempimento
0C      rp_TmpRas         ds.l     1        ;^Buffer intermedio
10      rp_AreaInfo       ds.l     1        ;^Struttura di Info
14      rp_GelsInfo       ds.l     1        ;^Struttura di GelInfo
18      rp_Mask           ds.b     1        ;Maschera di scrittura
19      rp_FgPen          ds.b     1        ;Pen primo piano
1A      rp_BgPen          ds.b     1        ;Pen sfondo
1B      rp_AOLPen         ds.b     1        ;Flood-Pen
1C      rp_DrawHode       ds.b     1        ;Modo grafico
1D      rp_AreaPtSz       ds.b     1        ;Parole di Flood
1E      rp_Dummy          ds.b     1        ;Dummy
1F      rp_linpatcnt      ds.b     1        ;Poly-Count
20      rp_Flags          ds.w     1        ;Uso sistema
22      rp_LinePtrn       ds.w     1        ;Righe
24      rp_cp_x           ds.w     1        ;Posizione del x Pen
26      rp_cP_y           ds.w     1        ;Posizione del y Pen
28      rp_minterms       ds.b     8        ;Controllo Blitter
30      rp_PenWidth       ds.w     1        ;Larghezza Pen
32      rp_PenHeight      ds.w     1        ;Altezza Pen
34      rp_Font           ds.l     1        ;^Font
38      rp_AlgoStyle      ds.b     1        ;Parametri del testo:
39      rp_TxFlags        ds.b     1
3A      rp_TxHeight       ds.w     1
3C      rp_TxWidth        ds.w     1
3E      rp_TxBaseline     ds.w     1
40      rp_TxSpacing      ds.w     1
42      rp_RP_User        ds.l     1        ;^Reply-Port
```

```
46      rp_vordreserved  ds.b     14
54      rp_longreserved  ds.b     8
5C      rp_reserved      ds.b     8
64      rp_SIZEOF        ds.w     0


00      ai_VctrTbl       ds.l     1
04      ai_VctrPtr       ds.l     1
08      ai_FlagTbl       ds.l     1
0C      ai_FlagPtr       ds.l     1
10      ai_Count         ds.w     1
12      ai_MaxCount      ds.w     1
14      al_FirstX        ds.w     1
16      ai_FlrstY        ds.w     1
18      ai_SIZEOF        ds.w     0

ONE_DOTn         equu     1
ONE_DOT          equ      $2
FRST_DOTn        equ      0
FRST_DOT         equ      1


;REGIONS
;-------

00      rg_bounds                 ds.b     ra_SIZEOF
08      rg_RegionRectangle        ds.l     1
0C      rg_SIZEOF                 ds.w     0

00      rr_Next          ds.l     1
04      rr_Prev          ds.l     1
08      rr_bounds        ds.b     ra_SIZEOF
10      rr_SIZEOF        ds.w     0


;Sprites
;-------
00      ss_posctldata    ds.l     1        ;^Dati dello Sprite
04      ss_height        ds.w     1        ;Altezza
06      ss_x             ds.w     1        ;Positione X attuale
08      ss_y             ds.w     1        ;Positione Y attuale
0A      ss_num           ds.w     1        ;Numero Sprite (0..7)
0C      ss_SIZEOF        ds.w     0


;Testo
;-----

FS_NORMAL        equ      0
FSB_EXTENDED     equ      3
FSF_EXTENDED     equ      8
FSB_ITALIC       equ      2
FSF_TALIC        equ      4
FSB_BOLD         equ      1
FSF_BOLD         equ      2
```

```
FSB_UNDERLINED   equ      0
FSF_UNDERLINED   equ      1


FPB_ROMFONT      equ      0
FPF_ROMFONT      equ      1
FPB_DISKFONT     equ      1
FPF_DISKFONT     equ      2
FPB_REVPATH      equ      2
FPF_REVPATH      equ      4
FPB_TALLDOT      equ      3
FPB_TALLDOT      equ      8
FPB_WIDEDOT      equ      4
FPF_WIDEDOT      equ      16
FPB_PROPORTIONAL equ      5
FPF_PROPORTIONAL equ      32
FPB_DESIGNED     equ      6
FPF_DESIGNED     equ      64
FPB_REMOVED      equ      7
FPF_REMOVED      equ      128



00       ta_Name          ds.l     1
04       ta_YSize         ds.w     1
06       ta_Style         ds.b     1
07       ta_Flags         ds.b     1
08       ta_SIZEOF        ds.w     0

14       tf_YSize         ds.w     1
16       tf_Style         ds.b     1
17       tf_Flags         ds.b     1
18       tf_XSize         ds.w     1
1A       tf_Baseline      ds.w     1
1C       tf_BoldSmear     ds.w     1
1E       tf_Accessors     ds.w     1
20       tf_LoChar        ds.b     1
21       tf_HiChar        ds.b     1
22       tf_CharData      ds.l     1
26       tf_Modulo        ds.w     1
28       tf_CharLoc       ds.l     1
2C       tf_CharSpace     ds.l     1
30       tf_CharKern      ds.l     1
34       tf_SIZEOF        ds.w     0

; View
;------

V_PFBA           equ $40
V_DUALPF         equ $400
V_HIRES          equ $8000
V_LAC            equ 4
V_HAM            equ $800
V_SPRITES        equ $4000
```

```
GENLOCK_VIDEO    equ      2

cm_Flags         ds.b     1
cm_Type          ds.b     1
cm_Count         ds.w     1
cm_ColorTable    ds.l     1
cm_SIZEOF        ds.w     0



; Struttura della ViewPort
;------------------------
00       vp_Next          ds.l     1        ;^Successivo
04       vp_ColorMap      ds.l     1
08       vp_DspIns        ds.l     1
0C       vp_SprIns        ds.l     1
10       vp_ClrIns        ds.l     1
14       vp_UCopIns       ds.l     1
18       vp_DWidth        ds.w     1        ;Larghezza
1A       vp_DHeight       ds.w     1        ;Altezza
1C       vp_DxOffset      ds.w     1
1E       vp_DyOffset      ds.w     1
20       vp_Modes         ds.w     1
22       vp_reserved      ds.w     1
24       vp_RasInfo       ds.l     1
28       vp_SIZEOF        ds.w     0

00       v_ViewPort       ds.l     1
04       v_L0FCprList     ds.l     1
08       v_SHFCprList     ds.l     1
0C       v_DyOffset       ds.w     1
0E       v_DxOffset       ds.w     1
10       v_Modes          ds.w     1
12       v_SIZEOF         ds.w     0

00       cp_collPtrs      ds.l     1
04       cp_SIZEOF        ds.w     0

00       ri_Next          ds.l     1
04       ri_BitMap        ds.l     1
08       ri_RxOffset      ds.w     1
0A       ri_RyOffset      ds.w     1
0C       ri_SIZEOF        ds.w     0
```

A4.5 Devices

```
;Importato da  exec:
;------------------
CMD_NONSTD       equ      9
IO_SIZE          equ      $20
IOSTD_SIZE       equ      $30
LN_SIZE          equ      $0E
MN_SIZE          equ      $14
```

```
TV_SIZE          equ      8
LIB_SIZE         equ      $22
HP_SIZE          equ      $22
pf_SIZEOF        equ      $E8
TC_SIZE          equ      $5C
LN_PRI           equ      9
;----------------------------



;Audio
;-----

ADHARD_CHANNELS           equ      4

ADALLOC_MINPREC           equ      -128
ADALLOC_MAXPREC           equ      127

CMD_NONSTD                equ      9

ADCMD_FREE                equ      9
ADCHD_SETPREC             equ      10
ADCMD_FINISH              equ      11
ADCMD_PERVOL              equ      12
ADCMD_LOCK                equ      13
ADCMD_WAITCYCLE           equ      14

ADCMDB_NOUNIT             equ      5
ADCMDF_NOUNIT             equ      32
ADCMD_ALLOCATE            equ      ADCMDF_NOUNIT

ADIOB_PERVOL              equ      4
ADIOF_PERVOL              equ      16
ADIOB_SYNCCYCLE           equ      5
ADIOF_SYNCCYCLE           equ      32
ADIOB_NOWAIT              equ      6
ADIOF_NOWAIT              equ      64
ADIOB_WRITEMESSAGE        equ      7
ADIOF_WRITEMESSAGE        equ      128

ADIOERR_NOALLOCATION      equ      -10
ADIOERR_ALLOCFAILED       equ      -11
ADIOERR_CHANNELSTOLEN     equ      -12

20       ioa AllocKey     ds.w     1
22       ioa_Data         ds.l     1
26       ioa_Length       ds.l     1
2A       ioa_Period       ds.w     1
2C       ioa_Volume       ds.w     1
2E       ioa_Cycles       ds.w     1
30       ioa_WriteMsg     ds.b     MN_SIZE
         ioa_SIZEOF       equ      $44
```

```
;bootblock
;---------

00      BB_ID           ds.b    4
04      BB_CHKSUM       ds.l    1
08      BB_DOSBLOCK     ds.l    1
        BB_ENTRY        equ     $0C
        BB_SIZE         equ     $0C

BOOTSECTS       equ     2

BBNAME_DOS      equ     444F5300        ; 'DOS'«8
BBNAME_KICK     equ     4B49434B        ; 'KICK'


;CLIPBOARD
;---------

CBERR_OBSOLETEID        equ     1

00      cu_Node         ds.b    LN_SIZE
0E      cu_UnitNum      ds.l    1

00      io_Message      ds.b    MN_SIZE
14      io_Device       ds.l    1
18      io_Unit         ds.l    1
1C      io_Command      ds.w    1
1E      io_Flags        ds.b    1
1F      io_Error        ds.b    1
20      io_Actual       ds.l    1
24      io_Length       ds.l    1
28      io_Data         ds.l    1
2C      io_Offset       ds.l    1
50      io_ClipID       ds.l    1
        iocr_SIZEOF     equ     $34

PRIMARY_CLIP    equ     0

00      sm_Msg                  ds.b    MN_SIZE
14      sm_Unit                 ds.w    1
16      sm_ClipID               ds.l    1
        satisfyMsg_SIZEOF       equ     $1A


;CONSOLE
;-------

CD_ASKKEYMAP    equ     9
CD_SETKEYMAP    equ     10

SGR_PRIMARY     equ     0
SGR_BOLD        equ     1
SGR_ITALIC      equ     3
SGR_UNDERSCORE  equ     4
```

```
SGR_NEGATIVE      equ       7
SGR_BLACK         equ       30
SGR_RED           equ       31
SGR_GREEN         equ       32
SGR_YELLOW        equ       33
SGR_BLUE          equ       34
SGR_MAGENTA       equ       35
SGR_CYAN          equ       36
SGR_WHITE         equ       37
SGR_DEFAULT       equ       39
SGR_BLACKBG       equ       40
SGR_REDBG         equ       41
SGR_GREENBG       equ       42
SGR_YELLOWBG      equ       43
SGR_BLUEBG        equ       44
SGR_MAGENTABG     equ       45
SGR_CYANBG        equ       46
SGR_WHITEBG       equ       47
SGR_DEFAULTBG     equ       49
SGR_CLR0          equ       30
SGR_CLR1          equ       31
SGR_CLR2          equ       32
SGR_CLR3          equ       33
SGR_CLR4          equ       34
SGR_CLR5          equ       35
SGR_CLR6          equ       36
SGR_CLR7          equ       37
SGR_CLR0BG        equ       40
SGR_CLR1BG        equ       41
SGR_CLR2BG        equ       42
SGR_CLR3BG        equ       43
SGR_CLR4BG        equ       44
SGR_CLR5BG        equ       45
SGR_CLR6BG        equ       46
SGR_CLR7BG        equ       47

DSR_CPR           equ       6

CTC_HSETTAB       equ       0
CTC_HCLRTAB       equ       2
CTC_HCLRTABSALL   equ       5

TBC_HCLRTAB       equ       0
TBC_HCLRTABSALL   equ       3

;gameport
;---------

GPD_READEVENT     equ       9
GPD_ASKCTYPE      equ       10
GPD_SETCTYPE      equ       11
GPD_ASKTRIGGER    equ       12
```

```
GPD_SETTRIGGER   equ      13
GPTB_DOWNKEYS    equ      0
GPTF_DOWNKEYS    equ      1
GPTB_UPKEYS      equ      1
GPTF_UPKEYS      equ      2

00       gpt_Keys          ds.w     1
02       gpt_Timeout       ds.w     1
04       gpt_XDelta        ds.w     1
06       gpt_YDelta        ds.w     1
         gpt_SIZEOF        equ      8

GPCT_ALLOCATED             equ      -1
GPCT_NOCONTROLLER          equ      0
GPCT_MOUSE                 equ      1
GPCT_RELJOYSTICK           equ      2
GPCT_ABSJOYSTICK           equ      3
GPDERR_SETCTYPE            equ      1


;input
;------

IND_ADDHANDLER   equ      9
IND_REMHANDLER   equ      10
IND_WRITEEVENT   equ      11
IND_SETTHRESH    equ      12
IND_SETPERIOD    equ      13
IND_SETMPORT     equ      14
IND_SETMTYPE     equ      15
IND_SETMTRIG     equ      16


;INPUTEVENT
;----------

IECLASS_NULL               equ      0
IECLASS_RAWKEY             equ      1
IECLASS_RAWMOUSE           equ      2
IECLASS_EVENT              equ      3
IECLASS_POINTERPOS         equ      4
IECLASS_TIMER              equ      6
IECLASS_GADGETDOWN         equ      7
IECLASS_GADGETUP           equ      8
IECLASS_REQUESTER          equ      9
IECLASS_MENULIST           equ      10
IECLASS_CLOSEWINDOW        equ      11
IECLASS_SIZEWINDOW         equ      12
IECLASS_REFRESHWINDOW      equ      13
IECLASS_NEWPREFS           equ      14
IECLASS_DISKREMOVED        equ      15
IECLASS_DISKINSERTED       equ      16
IECLASS_ACTIVEWINDOW       equ      17
IECLASS_INACTIVEWINDOW     equ      18
```

```
IECLASS_MAX                         equu      $12

IECODE_UP_PREFIX                    equ       $80
IECODEB_UP_PREFIX                   equ       7
IECODE_KEY_CD_FIRST                 equ       0
IECODE_KEY_CD_LAST                  equ       $77
IECODE_COMM_CD_FIRST                equ       $78
IECODE_COMM_CODE_LAST               equ       $7F

IECODE_C0_FIRST                     equ       $00
IECODE_C0_LAST                      equ       $1F
IECODE_ASCII_FIRST                  equ       $20
IECODE_ASCII_LAST                   equ       $7E
IECODE_ASCII_DEL                    equ       $7F
IECODE_C1_FIRST                     equ       $80
IECODE_C1_LAST                      equ       $9F
IECODE_LATIN1_FIRST                 equ       $A0
IECODE_LATIN1_LAST                  equ       $FF
IECODE_LBUTTON                      equ       $68
IECODE_RBUTTON                      equ       $69
IECODE_MBUTTON                      equ       $6A
IECODE_NOBUTTON                     equ       $FF

IECODE_NEWACTIVE                    equ       1
IECODE_REQSET                       equ       1
IECODE_REQCLEAR                     equ       0

IEQUALIFIER_LSHIFT                  equ       1
IEQUALIFIERB_LSHIFT                 equ       0
IEQUALIFIER_RSHIFT                  equ       2
IEQUALIFIERB_RSHIFT                 equ       1
IEQUALIFIER_CAPSLOCK                equ       4
IEQUALIFIERB_CAPSLOCK               equ       2
IEQUALIFIER_CONTROL                 equ       8
IEQUALIFIERB_CONTROL                equ       3
IEQUALIFIER_LALT                    equ       16
IEQUALIFIERB_LALT                   equ       4
IEQUALIFIER_RALT                    equ       32
IEQUALIFIERB_RALT                   equ       5
IEQUALIFIER_LCOMMAND                equ       64
IEQUALIFIERB_LCOMMAND               equ       6
IEQUALIFIER_RCOMMAND                equ       128
IEQUALIFIERB_RCOMMAND               equ       7
IEQUALIFIER_NUMERICPAD              equ       $0100
IEQUALIFIERB_NUMERICPAD             equ       8
IEQUALIFIER_REPEAT                  equ       $0200
IEQUALIFIERB_REPEAT                 equ       9
IEQUALIFIER_INTERRUPT               equ       $0400
IEQUALIFIERB_INTERRUPT              equ       10
IEQUALIFIER_MULTIBROADCAST          equ       $0800
IEQUALIFIERB_MULTIBROADCAST         equ       11
IEQUALIFIER_LBUTTON                 equ       $1000
```

```
IEQUALIFIERB_LBUTTON            equu    12
IEQUALIFIER_RBUTTON             equ     $2000
IEQUALIFIERB_RBUTTON            equ     13
IEQUALIFIER_MBUTTON             equ     $4000
IEQUALIFIERB_MBUTTON            equ     14
IEQUALIFIER_RELATIVEMOUSE       equ     $8000
IEQUALIFIERB_RELATIVEMOUSE      equ     15


00      ie_NextEvent     ds.l    1
04      ie_Class         ds.b    1
05      ie_SubClass      ds.b    1
06      ie_Code          ds.w    1
08      ie_Qualifier     ds.w    1
        ie_EventAddress  equ     $0A
0A      ie_X             ds.w    1
0c      ie_Y             ds.w    1

0E      ie_TimeStamp     ds.b    TV_SIZE
16      ie_SIZEOF        ds.w    0


;KEYBOARD
;--------

KBD_READEVENT            equ     9
KBD_READMATRIX           equ     10
KBD_ADDRESETHANDLER      equ     11
KBD_REMRESETHANDLER      equ     12
KBD_RESETHANDLERDONE     equ     13

;Keymap
;------

00      km_LoKeyMapTypes ds.l    1
04      km_LoKeyMap      ds.l    1
08      km_LoCapsable    ds.l    1
0C      km_LoRepeatable  ds.l    1
10      km_HiKeyMapTypes ds.l    1
14      km_HiKeyMap      ds.l    1
18      km_HiCapsable    ds.l    1
1C      km_HiRepeatable  ds.l    1
        km_SIZEOF        equ     $20


KCB_NOP         equ     7
KCF NOP         equ     $80

KC_NOQUAL       equ     0
KC_VANILLA      equ     7
KCF_SHIFT       equ     1
KCF_ALT         equ     2
KCB_CONTROL     equ     2
KCF_CONTROL     equ     4
KCB_DOWNUP      equ     3
```

```
KCF_DOWNUP        equ     8

KCB_STRING        equ     6
KCB_STRING        equ     64

;Narrator
;--------

DEFPITCH          equ     110
DEFRATE           equ     150
DEFVOL            equ     64
DEFFREQ           equ     22200
NATURALF0         equ     0
ROBOTICF0         equ     1
MALE              equ     0
FEMALE            equ     1
DEFSEX            equ     MALE
DEFMODE           equ     NATURALF0

MINRATE           equ     40
MAXRATE           equ     400
MINPITCH          equ     65
MAXPITCH          equ     320
MINFREQ           equ     5000
MAXFREQ           equ     28000
MINVOL            equ     0
MAXVOL            equ     64

ND_NotUsed        equ     -1
ND_NoMem          equ     -2
ND_NoAudLib       equ     -3
ND_MakeBad        equ     -4
ND_UnitErr        equ     -5
ND_Cantalloc      equ     -6
ND_Unimpl         equ     -7
ND_NoWrite        equ     -8
ND_Expunged       equ     -9
ND_PhonErr        equ     -20
ND_RateErr        equ     -21
ND_PitchErr       equ     -22
ND_SexErr         equ     -23
ND_ModeErr        equ     -24
ND_FreqErr        equ     -25
ND_VolErr         equ     -26


30      NDI_RATE          ds.w    1
32      NDI_PITCH         ds.w    1
34      NDI_MODE          ds.w    1
36      NDI_SEX           ds.w    1
38      NDI_CHMASKS       ds.l    1
3C      NDI_NUMMASKS      ds.w    1
3E      NDI_VOLUME        ds.w    1
```

```
40      NDI_SAMPFREQ     ds.w     1
42      NDI_MOUTHS       ds.b     1
43      NDI_CHANMASK     ds.b     1
44      NDI_NUMCHAN      ds.b     1
45      NDI_PAD          ds.b     1
        NDI_SIZE         equ      $46

46      MRB_WIDTH        ds.b     1
47      MRB_HEIGHT       ds.b     1
48      MRB_SHAPE        ds.b     1
49      HRB_PAD          ds.b     1
4A      MRB_SIZE         equ      $4A



;PARALLEL
;--------

ParErr_DevBusy           equ      1
ParErr_BufTooBig         equ      2
ParErr_InvParam          equ      3
ParErr_LineErr           equ      4
ParErr_NotOpen           equ      5
ParErr_PortReset         equ      6
ParErr_InitErr           equ      7

PDCMD_QUERY              equ      CMD_NONSTD
PDCMD_SETPARAMS          equ      CMD_NONSTD+1
Par_DEVFINISH            equ      10


PARB_SHARED              equ 5
PARF_SHARED              equ 32
PARB_RAD_BOOGIE          equ 3
PARF_RAD_BOOGIE          equ 8
PARB_EOFMODE             equ 1
PARF_EOFMODE             equ 2

IOPARB_QUEUED            equ 6
IOPARF_QUEUED            equ 128
IOPARB_ABORT             equ 5
IOPARF_ABORT             equ 32
IOPARB_ACTIVE            equ 4
IOPARF_ACTIVE            equ 16
IOPTB_RWDIR              equ 3
IOPTF_RWDIR              equ 8
IOPTB_PBUSY              equ 2
IOPTF_PBUSY              equ 4
IOPTB_PAPEROUT           equ 1
IOPTF_PAPEROUT           equ 2
IOPTB_PSEL               equ 0
IOPTF_PSEL               equ 1


00      PTERMARRAY_0     ds.l     1
```

```
04      PTERMARRAY_1      ds.l     1
08      PTERMARRAY_SIZE   ds.w     0

30      IO_PEXTFLAGS      ds.l     1
34      IO_PARSTATUS      ds.b     1
35      IO_PARFLAGS       ds.b     1
36      IO_PTERMARRAY     ds.b     PTERMARRAY_SIZE
3E      IOEXTPar_SIZE     equ      $3E



;Serial
;------

SER_CTL                   equ      $11130000
SER_DBAUD                 equ      9600


SDCMD_QUERY               equ      9
SDCMD_BREAK               equ      10
SDCMD_SETPARAMS           equ      CMD_NONSTD+2
SER_DEVFINISH             equ      11

SERB_XDISABLED            equ      7
SERF_XDISABLED            equ      128
SERB_EOFMODE              equ      6
SERF_EOFMODE              equ      64
SERB_SHARED               equ      5
SERF_SHARED               equ      52
SERB_RAD_BOOGIE           equ      4
SERF_RAD_BOOGIE           equ      16
SERB_QUEUEDBRK            equ      3
SERF_QUEUEDBRK            equ      8
SERB_7WIRE                equ      2
SERF_7WIRE                equ      4
SERB_PARTY_ODD            equ      1
SERF_PARTY_ODD            equ      2
SERB_PARTY_ON             equ      0
SERF_PARTY_ON             equ      1

IOSERB_QUEUED             equ      6
IOSERF_QUEUED             equ      64
IOSERB_ABORT              equ      5
IOSERF_ABORT              equ      32
IOSERB_ACTIVE             equ      4
IOSERF_ACTIVE             equ      16
IOSTB_XOFFREAD            equ      4
IOSTF_XOFFREAD            equ      16
IOSTB_XOFFWRITE           equ      3
IOSTF_XOFFWRITE           equ      8
IOSTB_READBREAK           equ      2
IOSTF_READBREAK           equ      4
IOSTB_WROTEBREAK          equ      1
IOSTF_WROTEBREAK          equ      2
```

```
IOSTB_OVERRUN              equ      0
IOSTF_OVERRUN              equ      1

00       TERMARRAY_0       ds.l     1
04       TERMARRAY_1       ds.l     1
         TERMARRAY_SIZE    equ      8

30       IO_CTLCHAR        ds.l     1
34       IO_RBUFLEN        ds.l     1
38       IO_EXTFLAGS       ds.l     1
3C       IO_BAUD           ds.l     1
40       IO_BRKTIME        ds.l     1
44       IO_TERMARRAY      ds.b     TERMARRAY_SIZE
4C       IO_READLEN        ds.b     1
4D       IO_WRITELEN       ds.b     1
4E       IO_STOPBITS       ds.b     1
4F       IO_SERFLAGS       ds.b     1
50       IO_STATUS         ds.w     1
         IOEXTSER_SIZE     equ      $52

SerErr_DevBusy             equ      1
SerErr_BaudMismatch        equ      2
SerErr_InvBaud             equ      3
SerErr_BufErr              equ      4
SerErr_InvParam            equ      5
SerErr_LineErr             equ      6
SerErr_NotOpen             equ      7
SerErr_PortReset           equ      8
SerErr_ParityErr           equ      9
SerErr_InitErr             equ      10
SerErr_TimerErr            equ      11
SerErr_BufOverflow         equ      12
SerErr_NoDSR               equ      13
SerErr_NoCTS               equ      14
SerErr_DetectedBreak       equ      15


;timer
;-----

UNIT_MICROHZ      equ      0
UNIT_VBLANK       equ      1

00       TV_SECS           ds.l     1
04       TV_MICRO          ds.l     1
         TV_SIZE           equ      8

20       IOTV_TIME         ds.b     TV_SIZE
28       IOTV_SIZE         equ      $20

TR_ADDREQUEST     equ      9
TR_GETSYSTIME     equ      10
TR_SETSYSTIME     equ      11
```

```
PRD_RAWWRITE     equ     9
PRD_PRTCOMMAND   equ     10
PRD_DUMPRPORT    equ     11



aRIS     equ     0
aRIN     equ     1
aIND     equ     2
aNEL     equ     3
aRI      equ     4

aSGR0    equ     5
aSGR3    equ     6
aSGR23   equ     7
aSGR4    equ     8
aSGR24   equ     9
aSGR1    equ     10
aSGR22   equ     11
aSFC     equ     12
aSBC     equ     13

aSHORP0  equ     14
aSHORP2  equ     15
aSHORP1  equ     16
aSHORP4  equ     17
aSHORP3  equ     18
aSHORP6  equ     19
aSHORP5  equ     20

aDEN6    equ     21
aDEN5    equ     22
aDEN4    equ     23
aDEN5    equ     24
aDEN2    equ     25
aDEN1    equ     26

aSUS2    equ     27
aSUS1    equ     28
aSUS4    equ     29
aSUS3    equ     30
aSUS0    equ     31
aPLU     equ     32
aPLD     equ     33

aFNT0    equ     34
aFNT1    equ     35
aFNT2    equ     36
aFNT3    equ     37
aFNT4    equ     38
aFNT5    equ     39
aFNT6    equ     40
aFNT7    equ     41
aFNT8    equ     42
```

```
aFNT9   equ     43
aFNT10  equ     44

aPROP2  equ     45
aPROPI  equ     46
aPROP0  equ     47
aTSS    equ     48

aJFY5   equ     49
aJFY7   equ     50
aJFY6   equ     51
aJFY0   equ     52
aJFY2   equ     53
aJFY5   equ     54

aVERP0  equ     55
aVERP1  equ     56
aSLPP   equ     57
aPERF   equ     58
aPERF0  equ     59

aLMS    equ     60
aRMS    equ     61
aTMS    equ     62
aBMS    equ     63
aSTBM   equ     64
aSLRM   equ     65
aCAM    equ     66

aHTS    equ     67
aVTS    equ     68
aTBC0   equ     69
aTBC3   equ     70
aTBC1   equ     71
aTBC4   equ     72
aTBCALL equ     73
aTBSALL equ     74
aEXTEND equ     75



20      io_PrtCommand    ds.w     1
22      io_Parm0         ds.b     1
23      io_Parm1         ds.b     1
24      io_Parm2         ds.b     1
25      io_Parm3         ds.b     1
        iopcr_SIZEOF     equ      $26

20      io_RastPort              ds.l      1
24      lo_ColorHap              ds.l      1
28      io_Modes                 ds.l      1
2C      io_SrcX                  ds.w      1
2E      io_SrcY                  ds.w      1
30      io_SrcWidth              ds.w      1
```

```
32       io_SrcHeight              ds.w      1
34       io_DestCols               ds.l      1
38       io_DestRows               ds.l      1
3C       io_Special                ds.w      1
         iodrpr_SIZEOF             equ       $3E


SPECIAL_MILCOLS           equ      1
SPECIAL_MILCOWS           equ      2
SPECIAL_FULLCOLS          equ      4
SPECIAL_FULLROWS          equ      8
SPECIAL_FRACCOLS          equ      16
SPECIAL_FRACROWS          equ      32

22       io_Segment                ds.l      1
26       io_ExecBase               ds.l      1
2A       io_CmdVectors             ds.l      1
2E       io_CmdBytes               ds.l      1
32       io_NumCommands            ds.w      1
         io_SIZEOF                 equ       $34


du_Flags         equ      LN_PRI

IOB_QUEUED       equ      4
IOF_QUEUED       equ      16
IOB_CURRENT      equ      5
IOF_CURRENT      equ      32
IOB_SERVICING    equ      6
IOF_SERVICING    equ      64
IOB_DONE         equ      7
IOF_DONE         equ      128


DUB_STOPPED      equ      0
DUF_STOPPED      equ      10

P_PRIORITY       equ      0
P_STKSIZE        equ      $800


PB_IOR0          equ      0
PF_IOR0          equ      1
PB_IOR1          equ      1
PF_IOR1          equ      2
PB_EXPUNGED      equ      7
PF_EXPUNGED      equ      128

34               pd_Unit            ds.b    MP SIZE
56               pd_PrinterSegment ds.l    1
5A               pd_PrinterType     ds.w    1
5C               pd_SegmentData     ds.l    1
60               pd_PrintBuf        ds.l    1
64               pd_PWrite          ds.l    1
68               pd_PbothReady      ds.l    1
```

```
PPCB_GFX         equ      0
PPCF_GFX         equ      1
PPCB_COLOR       equ      1
PPCF_COLOR       equ      2


PPC_BWALPHA      equ      0
PPC_BWGFX        equ      1
PPC_COLORGFX     equ      3

PCC_BW           equ      1
PCC_YMC          equ      2
PCC_YMC_BW       equ      3
PCC_YMCB         equ      4


00       ped_PrinterName  ds.l     1
04       ped_Init         ds.l     1
08       ped_Expunge      ds.l     1
0C       ped_Open         ds.l     1
10       ped_Close        ds.l     1
14       ped_PrinterClass ds.b     1
15       ped_ColorClass   ds.b     1
16       ped_MaxColumns   ds.b     1
17       ped_NumCharSets  ds.b     1
18       ped_NumRows      ds.w     1
1A       ped_MaxXDots     ds.l     1
1E       ped_MaxYDots     ds.l     1
22       ped_XDotsInch    ds.w     1
24       ped_YDotsInch    ds.w     1
26       ped_Commands     ds.l     1
2A       ped_DoSpecial    ds.l     1
2E       ped_Render       ds.l     1
32       ped_TimeoutSecs  ds.l     1
         ped_SIZEOF       equ      $36

00       ps_NextSegment   ds.l     1
04       ps_runAlert      ds.l     1
08       ps_Version       ds.w     1
0A       ps_Revision      ds.w     1
         ps_PED           equ      $0C


SPECIAL_ASPECT            equ      $80
SPECIAL_DENSITYMASK       equ      $F00
SPECIAL_DENSITY1          equ      $100
SPECIAL_DENSITY2          equ      $200
SPECIAL_DENSITY3          equ      $300
SPECIAL_DENSITY4          equ      $400

PDERR_CANCEL              equ      1
PDERR_NOTGRAPHICS         equ      2
PDERR_INVERTHAM           equ      3
PDERR_BADDIMENSION        equ      4
PDERR_DIMENSIONOVFLOW     equ      5
```

```
PDERR_INTERNALMEMORY      equ       6
PDERR_BUFFERMEHORY        equ       7

:TRACKDISK
;---------

NUMCYLS                   equ       80
MAXCYLS                   equ       NUMCYLS+20
NUMSECS                   equ       11
NUMHEADS                  equ       2
MAXRETRY                  equ       10
NUMTRACKS                 equ       NUMCYLS*NUMHEADS
NUMUNITS                  equ       4

TD_SECTOR                 equ       512
TD_SECSHIFT               equ       9

TDB_EXTCOM                equ       15
TDF_EXTCOM                equ       $8000

;Commands
TD_MOTOR                  equ      9
TD_SEEK                   equ      10
TD_FORMAT                 equ      11
TD_REMOVE                 equ      12
TD_CHANGENUM              equ      13
TD_CHANGESTATE            equ      14
TD_PROTSTATUS             equ      15
TD_LASTCOMM               equ      15


;Comandi Estesi ( con Blocco esteso di IORequest)
ETD_WRITE                 equ       3
ETD_READ                  equ       2
ETD_MOTOR                 equ       9
ETD_SEEK                  equ       10
ETD_FORMAT                equ       11
ETD_UPDATE                equ       4
ETD_CLEAR                 equ       5


;Estensione blocco di IORequest
30                IOTD_COUNT        ds.l     1
34                IOTD_SECLABEL     ds.l     1
                  IOTD_SIZE         equ      $38

TD_LABELSIZE              equ       16


;Error-Codes (in IOActual)
TDERR_NotSpecified        equ       20
TDERR_NoSecHdr            equ       21
TDERR_BadSecPreamble      equ       22
TDERR_BadSecID            equ       23
TDERR_BadHdrSum           equ       24
```

```
TDERR_BadSecSum          equu     25
TDERR_TooFewSecs         equ      26
TDERR_BadSecHdr          equ      27
TDERR_WriteProt          equ      28
TDERR_DiskChanged        equ      29
TDERR_SeekError          equ      30
TDERR_NoMem              equ      31
TDERR_BadUnitNum         equ      32
TDERR_BadDriveType       equ      33
TDERR_DriveInUse         equ      34
```

APPENDICE A5: CLI

-Introduzione al CLI
-Preparazione del dischetto di lavoro
-Consigli di Include
-Consigli e trucchi per il CLI

La presente appendice vuole mettere il lettore in grado di conoscere il CLI almeno per quanto necessario ad effettuare i primi passi nella programmazione in Assembler.
Il paragrafo relativo ai consigli e ai trucchi presuppone tuttavia una buona comprensione degli aspetti abbastanza complicati del CLI.

CLI significa Command Line Interpreter. In parole povere, noi forniamo dei comandi tramite la tastiera. Il CLI interpreta tali comandi e li esegue. Praticamente esso corrisponde al livello utente dei computer CP/M oppure MS-DOS, solo che il CLI è molto più capace di quanto non lo sia per es. l'MS-DOS.

HFS

Ogni DOS si preoccupa prima di tutto di file. Il file System decide come ordinare tali file sul disco e come ritrovarli.
L'Amiga ha un file System gerarchico, chiamato brevemente HFS.

L'HFS si occupa della disposizione di file, chiamati anche documenti. Il confronto con un ufficio è sempre molto calzante, se si ipotizza per es.:
- Il disco (dischetto o disco rigido) è l'ufficio
- Una directory è un armadio
- Una subdirectory è un cassetto di tale armadio
- Una sub-subdirectory é un cassetto nel cassetto
- Un documento può trovarsi in una posizione a piacere:

→ in mezzo all'ufficio (sul pavimento)
→ nell'armadio (non in un cassetto)
→ in un cassetto
→ in un cassetto che si trova in un cassetto

A differenza di quanto accade in un armadio, nel nostro caso è possibile inserire in un cassetto ulteriori cassetti, e in essi altri cassetti etc.. e tuttavia il cassetto interno non ha bisogno di essere più piccolo di quello che lo contiene, cioè: la dimensione non è definita. Sarà possibile quindi inserire in un cassetto dei file con lunghezza a piacere, finché il disco non sarà pieno.

Nomi di dischi

Come i dischi, anche i cassetti hanno un nome, attribuibile a piacere. Un nome di disco termina sempre con un due punti. E' possibile tuttavia rivolgersi ad un disco non solo con il suo nome (molti non ce l'hanno) ma anche con il nome dell'apparecchiatura. Per tali nomi, vale:

DF0: = Dischetto 0
DF1: = Dischetto1
JH0: = Disco rigido 0 (Janus Harddisk)

Esiste ancora un nome, che è

SYS:

SYS: identifica sempre il dischetto Boot. Se il dischetto Boot si chiama per es. WBENCH, sarà possibile chiamarlo con

WBENCH:
oppure DFO:
oppure SYS:.

Nomi di directory o di file

Un cassetto viene chiamato directory. Nei comandi CLI troviamo spesso anche l'abbreviazione DIR oppure D.

Non esiste nessuna differenza sostanziale da tenere presente al momento dell'attribuzione del nome. E' addirittura possibile che un file e la directory che lo contiene abbiano lo stesso nome. In linea di massima si riconosce tuttavia un nome di directory grazie alla barra che lo segue.

Nomi di percorso

Nella maggior parte dei casi le operazioni hanno effetto sempre su un file. Prendiamo il seguente esempio:

Sul dischetto nel Drive	DF0:
c'è una directory chiamata	ASSEMBLER
in essa c'è una directory	
chiamata	SOURCES
ed in essa un file chiamato	TEST.S

A questo punto è possibile rivolgersi al file TEST.S con

```
DF0:ASSEMBLER/SOURCES/TEST.S
```

Questa forma viene chiamata nome di percorso. Il sistema più sicuro per arrivare ad un file è sempre quello di usare il nome di percorso completo. Ciò richiede lunghe operazioni di battitura ma esistono dei metodi di abbreviazione che vedremo in seguito.

Comandi CLI

I comandi CLI vengono battuti e conclusi con un Return. I| comando CLI più semplice è il DIR. DIR fa apparire sullo schermo la directory corrente. Se non si indica nessun nome di percorso, tutti i comandi hanno effetto sempre sulla directory corrente, La visualizzazione di quale directory è quella corrente è ottenibile con CD.

CD (Change Directory) con un nome di file modifica la directory corrente. Battiamo per es. (sempre con Return alla fine):

Comando	Effetto
`CD df0:`	`Ci troviamo nel punto più alto (nella directory radice)`
`DIR`	`Visuailzzazione di tutti i file e di tutte le directory a questo livello`
`MAKEDIR xyz`	`Inserimento di una nuova directory chiamata xyz`
`CD xyz`	`Ci troviamo ora nella nuova directory`

Preparazione di un dischetto di lavoro

Facciamo prima di tutto una copia del dischetto Workbench (meglio due) e lavoriamo sempre con la copia. Dopo l’accensione e l'apertura del disco Workbench, dovremmo vedere una icona con il nome CLI. Se ciò non accade, chiamiamo “Preferences” (tramite un clickaggio) e dentro essa clickiamo “CLI ON”; quindi salviamo questo stato. A partire da questo momento, ad ogni accensione, dovrebbe essere visibile la icona CLI.

Il CLI viene fatto partire tramite un clickaggio. Esso appare nella sua propria Window, che di solito è troppo piccola.
Ingrandiamola per quasi tutto lo schermo, in questa occasione vi do ancora un consiglio: non programmare mai delle finestre aventi dimensioni uguali a quelle dello schermo. Potrebbe accadere che, a causa di oscillazioni del monitor o della corrente di alimentazione, alcune parti della finestra non siano più visibili.

Al fine di entrare immediatamente nel CLI al momento dell’accensione, sarà necessario modificare la Startup-Sequence. Si tratta di un file (testo puro) con comandi CLI, che

viene eseguito automaticamente ad ogni accensione. Come tutti questi file Batch (eseguibili anche con EXECUTE nome di file) anche questo si trova nella s-Directory. Con “CD s” entriamo in tale directory, carichiamo il file “Startup-Sequence” nell’Editor (ED. ved. Manuale), togliamo l'ultima riga che dovrebbe essere “endcli > nil:” e rimemorizziamo il file.

Usiamo ora il tasto di “panico” (Control più ambedue le *A* di Amiga conteporaneamente) e facciamo ripartire il sistema. Sarà molto difficile riuscire ad evitare l’impiego di questa combinazione di tasti durante gli sviluppi di programmi in Assembler.

Creazione di spazio

Spesso il dischetto è purtroppo molto pieno e l'Assemblatore, con i suoi accessori non ci sta. Nel caso di HiSoft (DEVPAC) il problema viene risolto in mannera eccellente (ved. manuale), negli altri casi dobbiamo essere noi a procurare lo spazio.
Andiamo nella directory C (CD c:) e battiamo DIR. Qui troveremo tutti i comandi CLI e li cancelleremo quasi tutti (DELETE Nome).
Avremo veramente bisogno solo di CD, COPY, DELETE, DIR. MAKEDIR e dell'Editor ED, nonché di EXECUTE, se lavoriamo con l'Assembler Metacomco. Nel caso in cui manchi qualcosa, sarà sempre possibile usare il disco completo (l'originale o una copia fatta in precedenza) oppure ricopiare in esso il programma.

Per il resto è possibile spostarsi senza problemi da una directory all'altra. Con “CD Name” entriamo in una “Dir”, se in tale “Dir” c'è un'altra “Dir”, useremo di nuovo CD. Con “CD /” è possible salire di un livello, con due "/" saliremo di due livelli. Con “CD :” torneremo di nuovo in cima, mentre CD da solo ci mostra dove ci troviamo in questo momento.

Tramite questi passaggi è possibile cancellare tutti i font (il font di sistema nella ROM oppure nella Kickstart-RAM non viene elencato). Nel caso in cui non si utilizzi un BridgeBoard oppure Sidecar, naturalmente sono superflui anche tutti i file nel raccoglitore PC. Se si dispone di una sola stampante, si necessiterà solo del suo gestore (Ved, nomi in Preferences),tutti gli altri gestori possono venire cancellati. Per quanto concerne i Map-files (gestori di tastiera) avremo bisogno solo di I (italiano) ed usa 0, se si utilizza il DEVPAC. Quindi, per evitare equivoci e rimanere sul sicuro, usiamo le seguenti istruzioni:

```
cd :
delete fonsts all
delete Utilities all ;ignorare un eventuale errore
delete #?1.info
delete empty all
delete trashcan all
delete clock
```

Copiatura di Tool in Assembler

A questo punto dovremo copiare l'Assemblatore e, se necessario,il Linker (ALINK) nonché il Debugger (MonAm) in caso di HiSoft nella directory C. Si usa la directory C in quanto l'Amiga cerca tutti i file di programma prima di tutto nella directory corrente, quindi in C. Il vantaggio di questo metodo è che dovunque ci si trovi, i programmi verranno sempre trovati.

Ipotizziamo ora che si stia lavorando con ASSEM ed ALINK e che si disponga di un solo Drive. Se vogliamo accedere ai file di Include dovremo procedere come segue: cancelliamo dalla copia di lavoro i file di cui sopra, o anche più, quindi battiamo:

```
delete ram:#?
copy c:assign to ram:
copy c:dir to ram:
copy c:cd to ram:
copy c:makedir to ram:
copy c:delete to ram:
assign c: ram:
```

Ora inseriamo il dischetto con ASSEM ed ALINK e battiamo:

```
cd df0:
copy c/assem to ram:
copy c/alink to ram:
makedir ram:include
copy include to ram: all
```

A questo punto introduciamo di nuovo il disco di lavoro e battiamo:

```
cd df0:
copy ram: assem to c
copy ram:alink to c
makedir include
copy ram:include to include all
```

Consigli per Include

Nel caso in cui non sia stato possibile fare entrare tutti i file Include nel disco RAM o che manchi spazio sul dischetto di destinazione, limitiamoci a quelli più importanti (DOS, Exec, Graphics e Intuition). Facciamo però attenzione: alcuni file hanno la caratteristica di caricarne altri. Proprio all'inizio del file troviamo una istruzione che è “IF NOT DEF Kenner INCLUDE filename Kenner SET 1”. Ciò significa,se la Label Kenner non è definita, carica il file e imposta la Label. Quindi, spesso, viene caricato un intero file Include, al fine di utilizzare di esso solo uno o due dati. Dal momento che il file Include di questo tipo carica altri file di Include, ci si ritroverà ben presto con file enormi l'uno vicino all'altro.

E' così che un mini programma per Assembler arriva ad avere 2000 righe o più cosa che procura un enorme spreco di tempo sul dischetto (con un disco rigido si andrebbe molto meglio). Per evitare ciò, sarebbe opportuno “tagliare su misura” i file di Include. Prendiamo il più importante (il file principale) xxxlib.i, cancelliamo le righe con "IF NOT DEF" e assembliamo. Sarà l'assemblatore a dirci cosa gli manca. Copiamoci tali definizioni dal file da “includere”.

Consigli per il CLI

Help: Se non si conosce esattamente un comando, è molto utile battere tale
comando senza parametri. In questo caso dovrebbe apparire "Usage" (istruzioni per l'uso). Un altro sistema per ottenere un Help è quello di lasciare uno spazio bianco dopo il comando, seguilo da un punto interrogativo.

Mini-Editor

Se si vuole creare velocemente un MAKE-file e non si vuole chiamare l'Editor per quelle poche righe che si scriveranno, si può usare

```
copy * to filename
```

* è il simbolo della finestra CLI corrente. Questo modo viene terminato con Control barra rovesciata(\). Con “copy to prt:" è possibile fare uscire sulla stampante, al fine di effettuare delle pre-impostazioni (tramite Escape) oppure semplicemente per brevi annotazioni.

Copiatura veloce

Nell'esempio di cui sopra abbiamo copiato tramite il RAM-Disk,cosa che viene effettuata molto più velocemente, rispetto al disco vero e proprio. Ciò è dovuto al fatto che il DOS mette a disposizione solo un piccolo Buffer, e che di conseguenza tratta i file a piccoli pezzetti per volta. Nel caso del RAM-Disk,invece, la velocità è circa quadrupla.

Se si devono copiare molti file, è utile memorizzare il comando COPY nel RAM-Disk. Se non lo si fa, il DOS caricherà tutte le volte dal dischetto il comando COPY dopo ogni file. Quindi scriviamo prima:

```
copy c:copy to ram:
```

dopodiché scriviamo per es.:

```
ram:copy Sorgente to Destinazione
```

Infatti per quanto concerne i listati del presente libro, io li ho preparati tutti in una directory e, quando giravano, NON li ho copiati nella directory “libro”. Infatti se si dà il comando RENAME invece di COPY, si ottiene lo stesso risultato molto più velocemente, e si ha anche il vantaggio di non dover cancellare il sorgente.

Workbench più CLI

Se si vuole essere in grado di decidere solo dopo l'accensione fra l'utilizzo di WB o di CLI, sarà possibile naturalmente partire con WB e solo in seguito clickare la icona CLI. Un'altra alternativa sarebbe la seguente: lasciare nel file Startup-Sequence l'ultima riga (endcli > nil:) esattamente come è in originale e scrivere sotto di essa un'altra riga:

```
NEWCLI CON:x/y/w/h/text
```

Per x/y/w/h dovremo inserire dei numeri, cioè:
x,y: angolo superiore sinistro della finestra
w,h: larghezza e altezza della finestra
Per "testo": un testo a piacere (che rientri nella larghezza)

Per cominciare poniamo la finestra in un angolo e lasciamola piccola. In caso di necessità la potremo sempre spostare con il Mouse e ne potremo modificare Ie dimensioni. In ogni caso sarà sufficiente clickare in tale finestra per trovarsi immediatamente nel CLI.

CLI veloce

Tutti i comandi CLI sono programmi che vengono caricati dal dischetto ad ogni chiamata. E' utile mettere i comandi più importanti nel RAM-Disk (oppure addirittura tutti).
Ciò ha luogo con

```
COPY nome comando to ram:c
```

A questo punto abbiamo due possibilità: una è quella di mettere sempre prima del comando il prefisso ram: l'altra è quella di dire al DOS una volta per tutte che dovrà cercare il comando nel RAM-Disk. Quest'ultima possibilità ha luogo tramite

```
assign c: ram:
```

I comandi corrispondenti possono anche venire scritti nella Startup-Sequence, per cui si otterrà automaticamente l'assign dopo l'accensione.

Se si vuole utilizzare il RAM-Disk anche sotto Workbench, sarà necessaria una icona di dischetto. Ciò viene generata (anche nella sequenza di Startup) dal semplice comando "dir ram:"

AMIGA
assembler

Peter
Wollschlaeger

Qualsiasi linguaggio di programmazione ad alto livello, come Basic o Pascal, ma anche il linguaggio C, impongono all'utente alcune limitazioni. Ciò può essere causato o dal fatto che determinate cose non sono sempre realizzabili oppure che i programmi funzionano troppo lentamente. È necessario dunque impadronirsi della possibilità di massimizzare l'utilizzo del proprio calcolatore per mezzo del suo linguaggio "naturale", il linguaggio Assembly. I comandi Assembly e le funzioni DOS vengono esemplificate con brevi programmi di difficoltà sempre crescente; la conoscenza necessaria per apprendere quanto trattato con i programmi viene data gradatamente analizzando i diversi casi.
Passo dopo passo i lettori impareranno quindi tutto quanto concerne il funzionamento interno del sistema operativo del proprio Amiga in modo tale da raggiungere risultati insperati nel campo della programmazione.

SOMMARIO

- Fondamenti del microprocessore 68000
- Programmazione di sistema con numerosi esempi
- Programmazione dell'Intuition
- Grafica a colori
- Tutte le routine di sistema con i parametri
- EXEC e multitasking

GRUPPO EDITORIALE JACKSON

ISBN 88-7056-955-1

9 788870 569551

L. 59.000 Cod. CL757